北京大学计算机系
教学用书

软件工程

——技术、方法与环境

[illegible]

内容简介

本书是在北京大学计算机科学技术系使用的软件工程讲义的基础上，由主讲、主考教师编写而成的，既是北京大学计算机系本科生指定教材，也是北京市高等教育自学考试指定教材。

本书结合国内外软件工程的发展，特别是国家“八五”攻关成果，详细地讲述了软件工程的基本内容，包括基本概念、基本模型、基本方法及相应的支持工具。本书注重基础知识的系统性，同时注意选材的先进性，内容全面、层次清楚。

出版社　：　北京大学出版社
系列名　：北京大学计算机系教学用书
作者　　：　王立福
出版日期：2002 年 1 月
国标编号：　7-301-03227-7/TP.318
条形码　：9787301032275
字数　　：350 千字
印张　　：14
开本　　：787*1092　1/16

目　录

第一章　软件工程概论

软件工程这一术语首次出现在1968年的NATO会议上。60年代以来，随着计算机的广泛应用，软件生产率、软件质量远远满足不了社会发展的需求，成为社会、经济发展的制约因素。当时，软件开发虽然有一些工具支持，例如编译连接器等，但基本上还是依赖开发人员的个人技能，没有可遵循的原理、原则和方法，也缺乏有效的管理。软件可靠性、可维护性较差，而且往往超出预期的开发时间要求。软件工程这一概念的提出，其目的是倡导以工程的原理、原则和方法进行软件开发，以期解决当时出现的“软件危机”。

软件危机是指在计算机软件开发和维护过程中所遇到的一系列问题，例如：

① 不能正确地估计软件开发成本和进度，致使实际开发成本往往高出预算很多；

② 软件产品不可靠，满足不了用户的需求，甚至无法使用；

③ 交付使用的软件不易演化，以致于人们不得不重复开发类似的软件；

④ 软件生产率低下，远远满足不了社会发展的需求；

……

产生软件危机的原因很多，除了与软件本身固有的特征有关以外，还与软件开发范型、软件设计方法、软件开发支持以及软件开发管理等有关。

软件工程作为一门学科已有近30年的历史，其发展大体可划分为两个时期。

60年代末到80年代初，软件系统的规模、复杂性以及在关键领域的广泛应用，促进了软件开发过程的管理及工程化开发。这一时期主要围绕软件项目，开展了有关开发模型、支持工具以及开发方法的研究。其主要成果体现为：提出了瀑布模型；开发了诸多结构化语言（例如PASCAL语言、C语言、Ada语言等）和结构化方法（例如“自顶向下”方法），试图向程序员提供好的需求分析和设计方法并开发了一些支持工具，例如调试工具等；开始出现各种管理方法，例如费用估算、文档复审；开发了一些相应支持工具，例如计划工具、配置管理工具等。这一时期的主要特征可概括为：前期主要研究系统实现技术，后期则开始强调管理及软件质量。

自“软件工厂”这一概念提出以来，80年代初主要围绕软件工程过程，开展了有关软件生产技术，特别是软件复用技术和软件生产管理的研究和实践。其主要成果是提出了具有广泛应用前景的面向对象方法和相关的语言（例如Smalltalk，C++，Eiffel等）；大力开展了计算机辅助软件工程（CASE）的研究与实践（例如我国在“七五”、“八五”期间，均把这一研究作为国家重点科技攻关项目），各类CASE产品相继问世。其间，最显著的事件是过程改进项目，该项目的目标是在工业实践中，建立一种量化的评估程序，判定软件组织成熟的程度。

近几年来，软件工程的研究已从过程（管理）转向产品（开发），更加注重新的程序开发范型和软件生产。其中，从大规模开发环境角度，开展了面向用户语言以及复用技术的研究；从抽象程序设计的角度，更加注重需求分析规格说明的形式化研究。与此同时，高智能、高自动化的CASE成为软件工程技术研究的热点。

1.1 软件工程概念

计算机系统中的程序及其文档称为软件。其中，程序是计算机任务的处理对象和处理规则的描述；文档是为了理解程序所需的阐述性资料。细言之，软件一词具有三层含义。一为个体含义，即指计算机系统中的程序及其文档；二为整体含义，即指在特定计算机系统中所有上述个体含义下的软件的总称，亦即计算机系统中硬件除外的所有成分；三为学科含义，即指在研究、开发、维护以及使用前述含义下的软件所涉及的理论、方法、技术所构成的学科。一般而言，工程是将科学论理和知识应用于实践的科学。在了解了“软件”和“工程”两个概念的基础上，软件工程可定义如下：

软件工程是一类求解软件的工程。它应用计算机科学、数学及管理科学等原理，借鉴传统工程的原则、方法，创建软件以达到提高质量，降低成本的目的。其中，计算机科学、数学用于构造模型与算法，工程科学用于制定规范、设计范型、评估成本及确定权衡，管理科学用于计划、资源、质量、成本等管理。软件工程是一门指导计算机软件开发和维护的工程学科。

1.2 软件工程框架

软件工程与其它工程(例如土木工程)一样，要有其自己的目标、活动和原则。软件工程的框架可概括为图 1.1 中所示的内容。

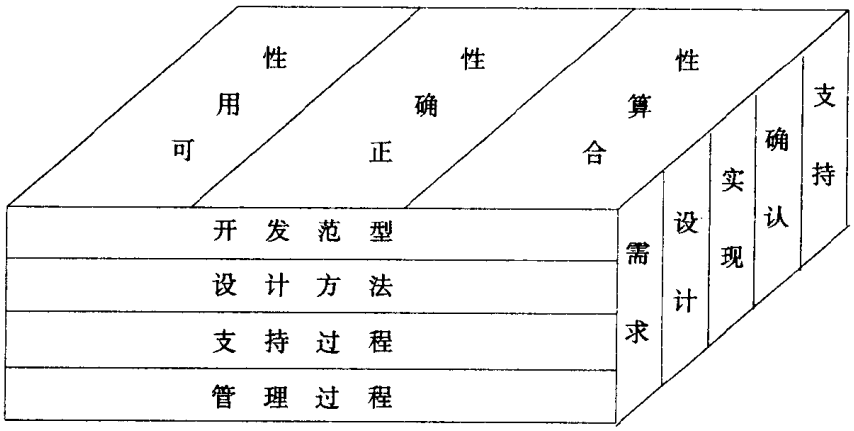


图1.1 软件工程框架

软件工程的目标可概括为“生产具有正确性、可用性以及开销合宜的产品”。正确性意指软件产品达到预期功能的程度。可用性意指软件基本结构、实现及文档为用户可用的程度。开销合宜是指软件开发、运行的整个开销满足用户要求的程度。这些目标的实现不论在理论上还是在实践中均存在很多问题有待解决，它们形成了对过程、过程模型及工程方法选取的约束。

软件工程活动是“生产一个最终满足需求且达到工程目标的软件产品所需要的步骤”。主要包括需求、设计、实现、确认以及支持等活动。需求活动包括问题分析和需求分析。问题分析获取需求定义，又称软件需求规约。需求分析生成功能规约。设计活动一般包括概要设计和详细设计。概要设计建立整个软件体系结构，包括子系统、模块以及相关层次的说明、每一模块的

接口定义。详细设计产生程序员可用的模块说明，包括每一模块中数据结构说明及加工描述。实现活动把设计结果转换为可执行的程序代码。确认活动贯穿于整个开发过程，实现完成后的确认，保证最终产品满足用户的要求。支持活动包括修改和完善。伴随以上活动，还有管理过程、支持过程、培训过程等。

围绕工程设计、工程支持以及工程管理，提出了以下四条基本原则：

第一条原则是选取适宜的开发模型。该原则与系统设计有关。在系统设计中，软件需求、硬件需求以及其它因素之间是相互制约、相互影响的，经常需要权衡。因此，必须认识需求定义的易变性，采用适宜的开发模型予以控制，以保证软件产品满足用户的要求。

第二条原则是采用合适的设计方法。在软件设计中，通常要考虑软件的模块化、抽象与信息隐蔽、局部化、一致性以及适应性等特征。合适的设计方法有助于这些特征的实现，以达到软件工程的目标。

第三条原则是提供高质量的工程支持。“工欲善其事，必先利其器”。在软件工程中，软件工具与环境对软件过程的支持颇为重要。软件工程项目的质量与开销直接取决于对软件工程所提供的支撑质量和效用。

第四条原则是重视开发过程的管理。软件工程的管理，直接影响可用资源的有效利用，生产满足目标的软件产品，提高软件组织的生产能力等问题。因此，仅当软件过程予以有效管理时，才能实现有效的软件工程。

综上所述，这一软件工程框架告诉我们，软件工程目标是可用性、正确性和合算性；实施一个软件工程要选取适宜的开发模型，要采用合适的设计方法，要提供高质量的工程支撑，要实行开发过程的有效管理；软件工程活动主要包括需求、设计、实现、确认和支持等活动，每一活动可根据特定的软件工程，采用合适的开发模型、设计方法、支持过程以及过程管理。根据软件工程这一框架，软件工程学科的研究内容主要包括：软件开发模型，软件开发方法，软件过程，软件工具，软件开发环境，计算机辅助软件工程(CASE)以及软件经济学等。

第二章　软件开发模型

软件开发模型是软件开发全部过程、活动和任务的结构框架。软件开发模型能清晰、直观地表达软件开发全部过程，明确规定要完成的主要活动和任务，它用来作为软件项目工作的基础。模型都应该是稳定和普遍适用的。

软件开发包括需求、设计、编码和测试等阶段，有时也包括维护阶段。软件开发模型对于不同的应用系统，允许采用不同的开发手段和方法，使用各种不同的程序设计语言以及各种不同技能的人员参与工作，还应允许采用不同的软件工具或各种不同的软件工程环境。

最早出现的软件开发模型是 1970 年 W. Royce 提出的瀑布模型，而后随着软件工程学科的发展和软件开发的实践，相继提出了演化模型、螺旋模型、增量模型、喷泉模型等。

2.1　瀑布模型

瀑布模型将软件生存周期的各项活动规定为依固定顺序连接的若干阶段工作，形如瀑布流水，最终得到软件产品。

瀑布模型可追溯到 50 年代末期，当时人们已感到必须先确认“做什么”，才能编制程序将其实现，即使是比较简单的小型问题也不例外。最简单的两级瀑布模型如图 2.1 所示。

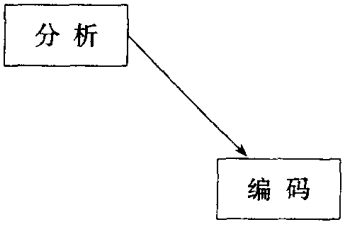


图2.1　两级瀑布模型

对于较大软件项目，问题更加复杂，两级模型已不能满足软件开发的实际需要，一个更精确的软件开发步骤可按需要解决问题的顺序依次为：做什么—如何做—制作—检测—使用，于是一个反映软件过程的基本框架如图 2.2 所示。

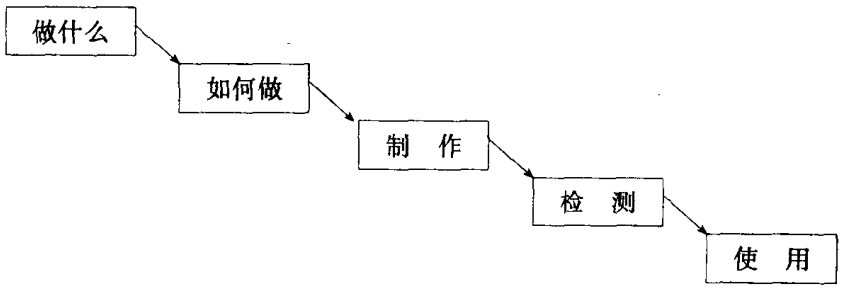


图2.2　瀑布模型雏型

该图表明，首先应给出软件的目标，确定要做什么；然后要决定如何达到这一目标，给出

策略、方法和步骤；继而加以实现，制作出所需要的软件；经过适当的检测，判定符合初始目标以后，方可投入运行和使用。可以说这是瀑布模型的雏型。

1970 年 W. Royce 首先将这一模型精确化，提出了具有多个开发阶段的瀑布模型，如图 2.3 所示。

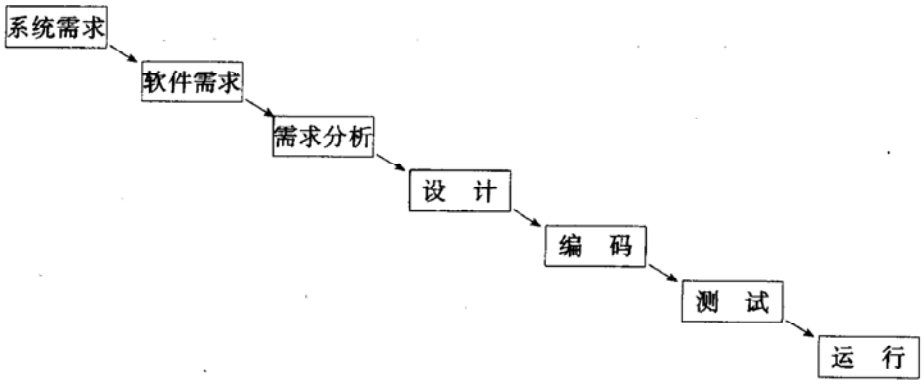


图 2.3 初始瀑布模型

这一模型规定了开发各阶段的活动为：提出系统需求、提出软件需求、需求分析、设计、编码、测试和运行，并且还规定了自上而下相互衔接的固定顺序，于是构成了人们熟知的瀑布模型。然而实践表明，各开发阶段间的关系并非完全是自上而下的线性图式，软件开发的实际情况是，每个开发阶段均具有以下特征：

① 从上一阶段接受本阶段工作的对象，作为输入；

② 对上述输入实施本阶段的活动；

③ 给出本阶段的工作成果，作为输出传入下一阶段；

④ 对本阶段工作进行评审，若本阶段工作得到确认，则继续下阶段工作，否则返回前一阶段，甚至更前阶段。

为表达向前阶段的反馈，在模型图中增加了虚线表示的箭头，构成了具有反馈回路的瀑布模型，如图 2.4 所示。

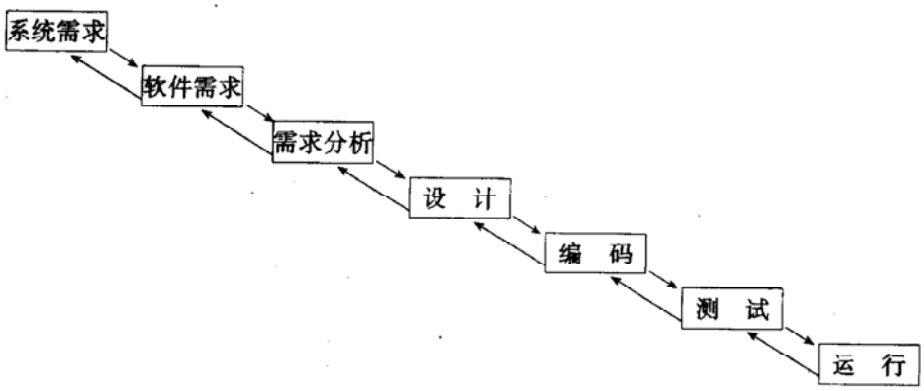


图 2.4 初始瀑布模型

瀑布模型有着不同形式的变种，比如，另一常见的具有反馈回路的瀑布模型包括的七个阶段是：可行性研究、需求分析和规约、设计和规约、编码和单元测试、集成测试和系统测试、交付、维护。不同形式瀑布模型的变种之间并无本质差别，选择哪一种形式可由软件项目特性及

开发组织决定。

许多采用瀑布模型的开发组织为有效地组织实施，制定了软件开发规范或开发标准。其中明确规定了各个开发阶段应交付的产品，这就为严格控制软件开发项目的进度，最终按时交付产品以及保证软件产品质量创造了有利条件。

瀑布模型20多年来之所以广泛流行，是因为它在支持结构化软件开发、控制软件开发的复杂性、促进软件开发工程化等方面起着显著作用。与此同时，瀑布模型在大量软件开发实践中也逐渐暴露出它的缺点。其中最为突出的缺点是该模型缺乏灵活性，无法通过开发活动澄清本来不够确切的软件需求，这些问题可能导致开发出的软件并不是用户真正需要的软件，无疑要进行返工或不得不在维护中纠正需求的偏差，为此必须付出高额的代价，为软件开发带来不必要的损失。并且，随着软件开发项目规模的日益庞大，该模型的不足所引发的问题显得更加严重。

2.2 演化模型

演化模型主要针对事先不能完整定义需求的软件开发。用户可以给出待开发系统的核心需求，并且当看到核心需求实现后，能够有效地提出反馈，以支持系统的最终设计和实现。软件开发人员根据用户的需求，首先开发核心系统。当该核心系统投入运行后，用户试用之，完成他们的工作，并提出精化系统、增强系统能力的需求。软件开发人员根据用户的反馈，实施开发的迭代过程。每一迭代过程均由需求、设计、编码、测试、集成等阶段组成，为整个系统增加一个可定义的、可管理的子集。如图2.5所示。

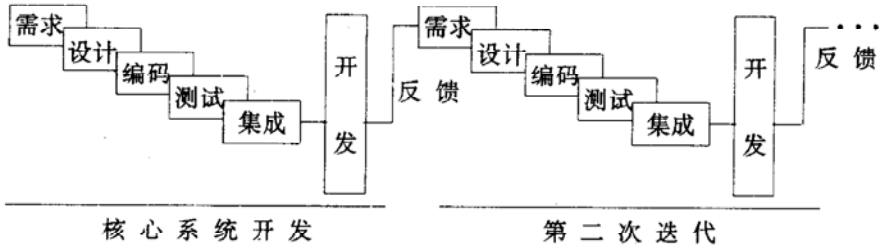


图2.5 演化模型

如果在一次迭代中，有的需求不能满足用户的要求，可在下一次迭代中予以修正。

演化模型在一定程度上减少了软件开发活动的盲目性。

2.3 螺旋模型

螺旋模型是在瀑布模型和演化模型的基础上，加入两者所忽略的风险分析所建立的一种软件开发模型。该模型于1988年由TRW公司B·鲍姆(BARRY W. BOEHM)提出。

软件风险是任何软件开发项目中普遍存在的问题，不同项目其风险有大有小。在制定软件开发计划时，系统分析员必须回答：项目的需求是什么，需要投入多少资源以及如何安排开发进度等一系列问题。然而若要他们当即给出准确无误的回答是不容易的，甚至几乎是不可能的。但系统分析员又不可能完全回避这一问题。凭借经验的估计给出初步的设想便难免带来

一定风险。实践表明，项目规模越大，问题越复杂，资源、成本、进度等因素的不确定性就越大，承担项目所冒的风险也越大。风险是软件开发不可忽视的潜在不利因素，它可能在不同程度上损害到软件开发过程和软件产品的质量。软件风险驾驭的目标是在造成危害之前，及时对风险进行识别、分析，采取对策，进而消除或减少风险的损害。

螺旋模型如图 2.6 所示。

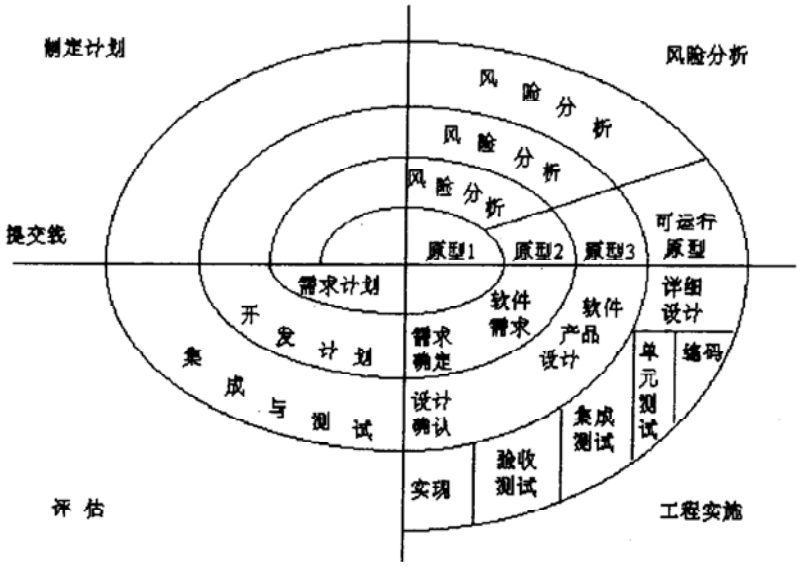


图 2.6 螺旋模型

沿着螺线旋转，在笛卡尔坐标的四个象限上分别表达了四个方面的活动，即：

① 制定计划——确定软件目标，选定实施方案，弄清项目开发的限制条件；

② 风险分析——分析所选方案，考虑如何识别和消除风险；

③ 实施工程——实施软件开发；

④ 客户评估——评价开发工作，提出修正建议。

沿螺线自内向外每旋转一圈便开发出更为完善的一个新的软件版本。例如，在第一圈，确定了初步的目标、方案和限制条件以后，转入右上象限，对风险进行识别和分析。如果风险分析表明，需求具有不确定性，那么在右下的工程象限内，所建的原型会帮助开发人员和客户，考虑其它开发模型，并把需求作进一步修正。

客户对工程成果作出评价后，给出修正建议。在此基础上需再次计划，并进行风险分析。在每一圈螺线上，风险分析的终点作出是否继续下去的判断。假如风险过大，开发者和用户无法承受，项目有可能终止。多数情况下沿螺线的活动会继续下去，自内向外逐步延伸，最终得到所期望的系统。图 2.7 给出了螺旋模型的另一图示。

如果对所开发项目的需求已有了较好的理解或较大的把握，无需开发原型，便可采用普通的瀑布模型。这在螺旋模型中可认为是单圈螺线。与此相反，如果对所开发项目的需求理解较差，需要开发原型，甚至需要不止一个原型的帮助，那就要经历多圈螺线。在这种情况下，外圈的开发包含了更多的活动。也可能某些开发采用了不同的模型。

螺旋模型适合于大型软件的开发，它是颇为实际的方法，它吸收了 T. Gilb 提出的软件工程“演化”概念。使得开发人员和客户对每个演化层出现的风险均有所了解，并继而作出反应。

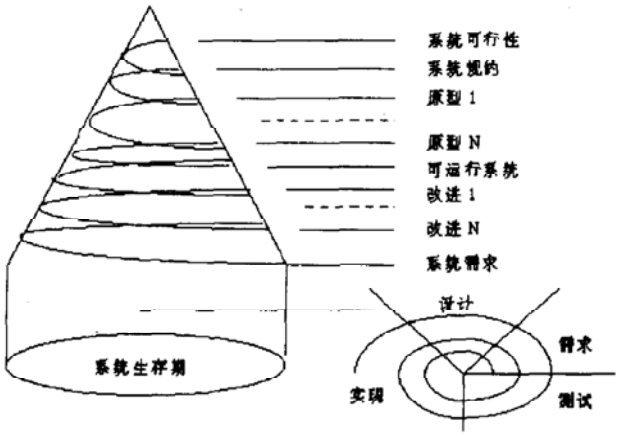


图 2.7 螺旋模型的另一种表示

和其它模型相比，螺旋模型的优越性较为明显，但要求许多客户接受和相信演化方法并不容易。本模型的使用需要具有相当丰富的风险评估经验和专门知识。如果项目风险较大，又未能及时发现，势必造成重大损失。此外，螺旋模型是出现较晚的新模型，远不如瀑布模型普及，要让广大软件人员和用户接受，还有待于更多的实践。

2.4 喷泉模型

喷泉模型体现了软件创建所固有的迭代和无间隙的特征。喷泉模型如图 2.8 所示。

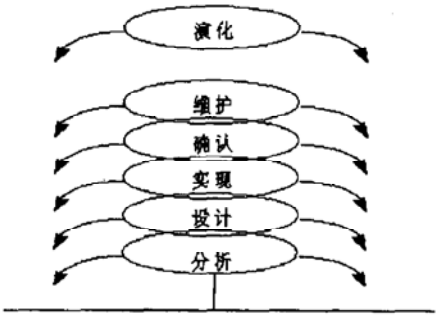


图 2.8 喷泉模型

这一模型表明了软件刻画活动需要多次重复。例如，在编码之前(实践之后)，再次进行分析和设计，其间，添加有关功能，使系统得以演化。同时，该模型还表明活动之间没有明显的间隙，例如在分析和设计之间没有明显的界限。

喷泉模型主要用于支持面向对象开发过程。由于对象概念的引入，使分析、设计、实现之间的表达没有明显间隙。并且，这一表达自然地支持复用。

2.5 增量模型

增量模型表达了如下所述的对开发活动的组织：

在设计了软件系统整体体系结构之后，首先完整地开发系统的一个初始子集；继之，根据这一子集，建造一个更加精细的版本。如此不断地进行系统的增量开发。这一模型如图 2.9 所示。

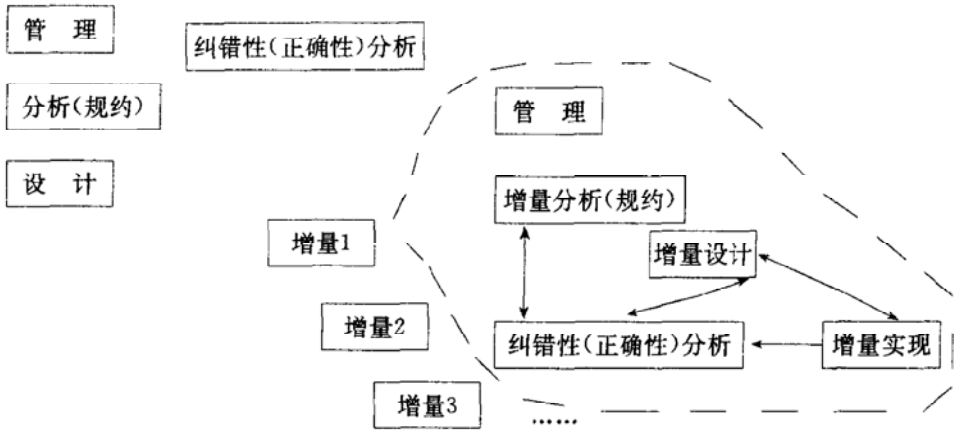


图 2.9 增量模型

该模型提供了一种在精化软件产品过程中体现用户经验的途径，也提供了一种周期性地进行最新软件维护、服务于各自用户的途径。

由于增量地进行开发，因此一个增量功能就比较容易理解和测试。这一模型广泛地使用于计算机工业中。

软件开发模型除以上介绍的几种模型外，还有原型/模拟模型、组装可复用构件模型等。

第三章 需求分析

需求分析阶段位于软件开发的前期，它的基本任务是准确地定义未来系统的目标，确定为了满足用户的需要系统必须做什么。

通常，人们在开始做任何一件事情以前，都必须对所要达到的目标或所要解决的问题有一个清楚而明确的认识。同样，软件开发人员在设计一个系统之前，必须事先了解用户希望未来的系统能做什么，这项工作通常是由称为系统分析员（或简称分析员）的人来承担的。需求分析实际上分为两个阶段——需求获取阶段和需求规约阶段。首先，系统分析员必须通过学习以及同用户的交互，熟悉用户领域的知识，并获得用户对未来系统的需求，这称为需求获取；其次，系统分析员在获得了用户的初步需求之后，必须进行一致性分析和检查，通过和用户协商解决其中存在的二义性和不一致性，并以一种规范的形式准确地表达用户的需求，形成所谓的需求规格说明书，这称为软件需求规约。

3.1 需求获取

开发系统总是有目的的。当用户要求我们开发一个软件系统时，他通常会提出一些对系统做什么的要求——用新系统代替现存系统或改进现存系统的工作模式。需求就是对系统特征以及为了完成用户任务，系统必须能够做什么的一个描述，它关心的是系统目标而不是系统实现。例如，假设我们正在为用户开发一个工资管理系统，其中的一个需求可能是每个月发一次工资，另一个需求可能是把达到或高于某个工资水平的雇员的工资直接存入银行，用户可能还会提出远程访问公司的工资管理系统的要求。所有这些需求都是对未来系统的功能或特征的一些特定描述。

需求获取的目的是清楚地理解所要解决的问题、完整地获取用户需求，主要包括以下几方面的活动：通过学习、请教领域专家、向用户提问等手段，了解所要解决的问题，理解用户的需要，确认谁是真正的用户，以及系统实现所受到的各种限制。

需求获取通常面临着三大挑战：

（1）问题空间理解

随着计算机在社会各方面不断广泛和深入的应用，大多数情况下，我们根本不可能在用户业务和应用方面以行家自居，但开发系统的任务又要求我们必须把握和深入理解问题空间，甚至比那些整天从事用户业务却没有全面考虑的人更要体察入微，而且必须尽可能快地做到这一点。

（2）人与人之间的通信

分析的挑战还包括通信，分析员在整个分析过程中都需要通信，为了从用户处获得对问题空间的理解和需求，分析员必须与用户通信，他要考虑通信问题而且要自行推敲，还要和同事互相切磋，最后还必须把他对问题空间的理解及由此得出的需求反馈给用户，检验他对需求的

理解。他可能还要帮助用户放弃一些受财力和进度限制而难以达到的需求。

(3) 需求的不断变化

需求一直处于不断的变化之中，管理人员或用户可以在某一特定的时刻人为冻结需求。但是，真正的需求和用户所需要的系统却不断在演变，影响需求变化的因素很多：用户、竞争者、协调人员、审批人员和技术人员。正如Gerhard Fisher于1989年指出的那样：我们不得不接受变化着的需求这个现实生活中的事实，而不应指责它是混乱思想的产物。作为分析员，应千方百计构造一些符号和策略以使他们的工作能适应变化。

下面我们分别讨论需求获取包含的内容、应遵循的原则以及采用的技术。

3.1.1 需求获取的内容

需求定义描述了未来系统的行为，我们可以把任何一个系统想象为一个有限状态自动机，即在任一时刻都处于一组可接受的状态集中。系统的活动，比如和输入/输出设备的交互、某一时刻时钟事件的到来等，都促使系统从一个状态变迁到另一个状态。需求定义就描述了系统可能处于的状态集合以及从一个状态到另一个状态的变迁规则。

依此观点我们可以把用户需求分为两大类：功能性需求和非功能性需求。前者定义了系统做什么，包括系统的所有输入、输出以及如何从输入映射到输出；后者定义了系统工作时的特性，例如系统对效率、可靠性、安全性、可维护性、可移植性、吞吐量以及符合某种标准等的要求，这些要求对我们选择解决问题的方案起限制作用。

我们可以进一步把以上两类用户需求分为以下几个方面：

1. 物理环境

- 操作设备的地点在何处?
- 位置是集中的还是分散的?
- 有无对环境的限制，诸如温度、湿度或磁场干扰?

2. 界面

- 有来自其它系统的输入吗?
- 有到其它系统的输出吗?
- 对数据格式有规定吗?
- 对数据存储介质有规定吗?

3. 用户或人的因素

- 谁将使用该系统?
- 存在多种类型的用户吗?
- 每种类型的用户，熟练程度如何?
- 每种类型的用户，需要接受什么样的训练?
- 对用户来说，理解和使用系统的难度如何?
- 用户错误操作系统的可能性如何?

4. 功能

- 系统将做什么?
- 系统何时做什么?

- 系统何时及如何修改或升级?
- 对于执行速度、响应时间或吞吐量有无限制?

5. 文档

- 需要哪些文档?
- 文档针对什么样的读者?

6. 数据

- 输入和输出数据的格式如何?
- 接收和发送数据的频率如何?
- 数据的准确性如何?
- 计算必须达到的精度如何?
- 系统处理的数据流量多少?
- 数据必须保持一段时间吗?

7. 资源

- 建造、使用和维护系统需要什么设备、人员或其它资源?
- 开发者必须具有什么样的技能?
- 系统将占用多大的物理空间?
- 对电力、暖气或空调的要求如何?
- 开发有规定的时间表吗?
- 用于开发的软硬件投资有无限制?

8. 安全性

- 必须对访问系统或系统信息加以控制吗?
- 一个用户的数据如何同其它用户的数据隔离开来?
- 用户程序如何同其它程序和操作系统隔离开来?
- 多长时间需要对系统做一次备份?
- 备份的数据必须放到另外的地方吗?
- 需要防火或防盗吗?

9. 质量保证

- 对系统的可靠性要求如何?
- 系统必须监测和隔离错误吗?
- 规定的系统平均出错时间是多少?
- 发生错误后,重启系统允许的最大时间是多少?
- 系统变化将如何反映到设计中?
- 维护只是包括修改错误,还是也包括对系统的改进?
- 在资源使用和时间响应上采取了哪些行之有效的方法?
- 系统的可移植性如何?

3.1.2 需求获取应遵循的原则

现实世界中的事物是复杂多变的,人们常常无法一下子认识清楚,这就需要采取一些符合

人类思维的方法和策略，其中，抽象和分解就是人们在认识世界和改造世界的长期实践中总结出来的行之有效的法则。反映到需求获取过程中，划分、抽象和投影是人们常用的组织信息的三条基本原则。

划分捕获问题空间的“整体/部分”关系。例如，让我们考察一个导弹防御系统，整个系统从功能上可以划分为目标监测、通信、情报收集与决策、防御武器发射和防御武器制导等子系统，于是处理一个复杂问题就变成了处理各个子问题，从而降低了问题的复杂性。进一步，各子问题如果仍然比较复杂，可以进一步把它们划分为子/子问题。

抽象捕获问题空间的“一般/特殊”或“特例”关系。例如，还是上面的导弹防御系统，我们可以把敌方的导弹分为不同的类型：常规导弹和核导弹；弹道导弹和可制导导弹；远程、中程和近程导弹，以上三个抽象的例子表明各种类型的导弹都是某种特殊的导弹或导弹的一个特例。

投影捕获问题空间的多维“视图”，让我们继续考察上面的例子，从不同对象的角度看待系统中的导弹，包括导弹操作员、指挥员、情报分析专家以及敌方的导弹，从每个角度对系统的观察反映的都是整个系统的一个“视图”，就像建筑设计师可以从正面、侧面、背面和剖面考察一座建筑物一样。

总之，划分、抽象和投影每个都定义了问题空间中的结构或层次关系。在不同的上下文中，A 从属于 B 意味着不同的含义。对于划分，意味着 A 是 B 的一部分；对于抽象，意味着 A 是 B 的一个特例，并继承了 B 的所有属性；对于投影，意味着 A 是 B 的一个视图。

3.1.3 需求获取采用的技术

许多分析员在获取用户需求时，经常是随意性地问一些主要的问题，记下回答，然后继续这样问下去。这种不规范的做法常常导致分析员和用户的思路前后不连贯，效率低下。在需求获取时，采用比较正规的方法的主要好处是可以简化任务。

作为一个好的需求获取技术的显著特征是：

① 方便通信(可以通过易于理解的语言)；

② 提供定义系统边界的方法；

③ 提供定义划分、抽象和投影的方法；

④ 鼓励分析员用问题空间的术语而不是软件术语去思考问题和编制文档；

⑤ 允许并提醒分析员有多种可供选择的设计方案；

⑥ 适应需求的变化。

下面我们以 Ivar Jacobson 提出的 use-case 驱动方法为例，介绍一种比较好的需求获取技术。用户在同系统的一次交互过程中所实施的一组行为上相关的动作序列称为一个 use-case，每个 use-case 描述了用户使用系统的一种特定方式，用户对系统提出的所有功能要求均以 use-case 表达。考虑电话交换系统，其中一个典型的 use-case 是用户打电话的过程，在这个过程中所发生的事件序列如图 3.1 所示。

对所有需要同系统交换信息的用户(或设备)标识其 use-case，所有 use-case 的集合描述了完整的系统功能。

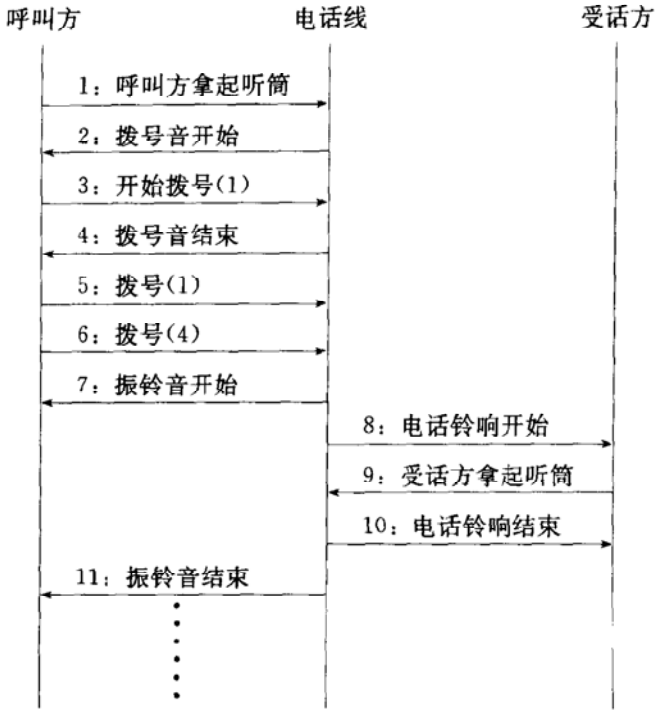


图 3.1　电话交换系统一个典型的 use-case

3.2　结构化分析方法

经过需求获取阶段的工作，系统分析员已经比较全面地获取了用户的需求，并形成需求定义，现在是进入到需求规约阶段的时候了。需求规约的主要目的是对需求定义进行分析，解决其中存在的二义性和不一致性，并以一种系统化的形式准确地表达用户的需求，形成所谓的需求规格说明书。本章我们主要介绍结构化分析方法，在本书的后面我们还将介绍面向对象的并发方法。

结构化方法是由 Edward Yourdon，Tom DeMarco 等人于 70 年代中后期提出的一种系统化开发软件的方法，该方法基于模块化的思想，采用“自顶向下，逐步求精”的技术对系统进行划分。分解和抽象是它的两个基本手段。结构化方法是结构化分析、结构化设计和结构化编程的总称。

结构化分析方法最初把整个系统表示成一张环境总图，标出系统边界及所有的输入、输出；逐步对系统进行细化，每细化一次，就把一些复杂的功能分解成较简单的功能，并增加细节描述；继续这种细化，直到所有的功能都足够简单，不需要再继续细化为止。结构化方法由于其简单易懂、容易使用，且出现较早，所以获得了广泛的应用。

3.2.1　模型表示

结构化分析方法把任何软件系统都视作一个数据变换装置，它接受各种形式的输入，通过变换产生各种形式的输出。数据流图(data-flow diagram，有时简称 DFD)就是一种描述数据变换的图形工具，是结构化分析方法最普遍采用的表示手段，但数据流图并不是结构化分析模型

的全部，数据字典和小说明为数据流图提供了补充，并用以验证图形表示的正确性、一致性和完整性，三者共同构成了结构化分析的模型。

1. 数据流图

数据流图是一种描述数据变换的图形工具，系统接受输入的数据，经过一系列的变换（或称加工），最后输出结果数据。对于比较大型的软件系统，把整个系统画在一张图中，不仅显得凌乱，而且层次不清、难以理解，有必要把数据流图分成多层。下面我们通过一个简单的例子对数据流图的成分加以说明。

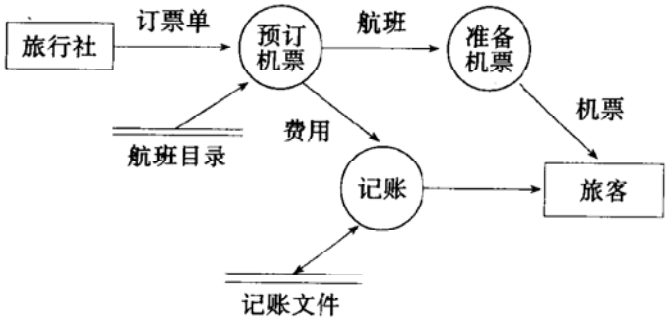


图 3.2　一个飞机票预订系统的数据流图

从图 3.2 我们可以看出，数据流图由以下四个基本成分组成：

（1）加工（用圆圈表示）

加工是对数据进行处理的单元，它接受一定的输入数据，对其进行处理，并产生输出，如图 3.2 中的预订机票、准备机票、记账等都是加工的例子。

（2）数据流（用箭头表示）

数据流表示数据和数据流向，如图 3.2 中的订票单、航班、费用、账单、机票等都是数据流的例子。

（3）数据存储（用两条平行线表示）

数据存储用于表示信息的静态存储，如图 3.2 中的航班目录、记账文件等就是数据存储的例子。

（4）数据源和数据潭（用矩形表示）

数据源和数据潭表示系统和环境的接口，是系统之外的实体，可以是人、物或其它软件系统。其中，数据源是数据流的起点，如图 3.2 中的“旅行社”就是数据源；数据潭是数据流的最终目的地，如图 3.2 中的“旅客”就是数据潭。

下面我们稍微详细地介绍一下各成分的作用及命名时的一些注意事项。

（1）加工

在分层的数据流图中，要对加工进行编号，以便于管理。加工也要选取适当的名字，以提高数据流图的可读性。下面给出一些加工的命名原则：

① 顶层的加工名就是软件项目的名字；

② 加工的名字最好使用动宾词组，如“计算费用”、“准备机票”、“检查合法性”、“产生工资单”等；也可以用主谓词组，如把上述的“计算费用”写成“费用计算”、“准备机票”写成“机票准备”等；

③ 不要使用意义空洞的动词作为加工名，如“计算”、“处理”、“加工”等，这样的名字没有给读者提供任何有意义的信息。

(2) 数据流

数据流表示数据和数据流向，通常由一组数据项组成，例如，数据流“订票单”由姓名、住址、电话、航班号、日期、始点、终点等数据项组成，而数据流“航班”则由航班号、日期、机型等数据项组成。

数据流可以从加工流向加工，在图 3.2 中，数据流“航班”是从加工“预订机票”流向加工“准备机票”的，数据流“费用”是从加工“预订机票”流向加工“记账”的；数据流也可以从数据源流向加工，或从加工流向数据潭，在图 3.2 中，数据流“订票单”是从数据源“旅行社”流向加工“预订机票”的，数据流“账单”是从加工“记账”流向数据潭“旅客”的；数据流还可以从加工流向数据存储，或从数据存储流向加工(一般流入或流出数据存储的数据流不需要标出名字，有数据存储的名字就可以了)，在图 3.2 中，数据存储“航班目录”和加工“预订机票”之间的数据流，数据存储“记账文件”和加工“记账”之间的数据流就是这样的例子。

两个加工之间可以有多个数据流，这些数据流之间没有任何联系，数据流图也不表明它们的先后次序。

每个数据流要有一个合适的名字，一方面为了区别不同的数据流，另一方面能使人容易理解数据流的意义。下面给出一些数据流命名的方法和注意事项：

① 数据流的名字用名词，或名词词组；

② 数据流模型是现实系统的抽象，需求获取时已经对现实系统有了比较清楚的认识，这就为数据流命名提供了直接的参考依据，命名时应尽量使用现实系统中已有的名字；

③ 把现实环境中传递的一组数据(这组数据组成一个数据流)中最重要的那个数据的名字作为数据流的名字；

④ 不要把控制流作为数据流；

⑤ 不要使用意义空洞的名词作为数据流名，如“数据”、“信息”等，这样的名字没有给读者提供任何有意义的信息。

(3) 数据存储

① 在分层数据流图中，数据存储一般局限在某一层或某几层，是和不同的抽象层次相关的；

② 数据存储的命名方法同数据流的命名方法相似。

(4) 数据源和数据潭

数据源和数据潭是表示系统和环境的接口，是系统之外的实体，命名时应符合环境的真实情况。

2. 数据字典

数据字典以一种准确的和无二义的方式定义所有被加工引用的数据流和数据存储，通常包括三类内容：数据流条目、数据存储条目和数据项条目。数据字典一般是按照一定的次序排列起来的，便于人们查阅。数据字典说明中经常使用一些常用的逻辑操作符(如表 3.1 所示)，以准确地表达被说明数据的结构。

表 3.1

操作符	含义描述
=	等价于(定义为)
+	与(顺序结构)
{ }	重复(循环结构)
[|]	或(选择结构)
( )	任选
m..n	界域

(1) 数据流条目

一个完整的数据流条目应该包括以下内容：

名称
描述
频率和数据量
数据结构

例如，本章后面“图书管理系统”例子中的“入库单”是一个数据流，对它的说明如下：

入库单＝分类目录号＋数量＋书名＋作者＋内容摘要＋价格＋购书日期

(2) 数据存储条目

名称
描述
数据存储方式
关键码
频率和数据量
安全性要求
数据结构

例如，同样为本章后面例子中的“目录文件”是一个数据存储，对它的说明如下：

文件名：　目录文件

组成：　{分类目录号＋书名＋作者＋内容摘要＋价格＋入库日期＋总数＋库存数＋{图书流水号}}

组织：　按分类目录号的字母顺序排列

(3) 数据项条目

名称
描述
数据类型
取值范围及缺省值
精度
计量单位

数据项条目的格式详见本章后面“需求规格说明书”中第 3.2.2 节“数据项说明”。

3. 小说明

小说明是用来描述加工的，在一个分层的数据流图中，上层的加工通过细化分解为下层的

更具体的加工。原则上，只要说明了最底层的基本加工，就可以理解上层的加工，对上层加工的说明是冗余的，所以小说明中可以只描述基本加工。当然，如果为了更便于理解，也可以在小说明中包括对上层加工的描述、总结和概括下层加工的功能。

小说明集中描述一个加工"做什么"，即加工逻辑，也包括其它一些和加工有关的信息，如执行条件、优先级、执行频率、出错处理等。加工逻辑是指用户对这个加工的逻辑要求，即这个加工的输入数据和输出数据的逻辑关系。小说明并不描述具体的加工过程。人们正在研究用来描述这种加工逻辑而不是加工过程的形式语言，遗憾的是目前尚无理想的结果。所以，目前小说明一般还是用自然语言、结构化自然语言、判定表和判定树等来描述。

小说明一般是按照加工的编号次序排列，当然也可以按照其它任何使用者觉得方便的次序排列，如按照加工名的字典顺序排列，特别是在计算机辅助支持下，小说明可依使用者的要求有多种排序方式。

(1) 结构化自然语言

结构化自然语言是介于形式语言和自然语言之间的一种语言。它虽然没有形式语言那样严格，但具有自然语言简单易懂的特点，同时又避免了自然语言结构松散的缺点。

结构化自然语言的语法通常分为内外两层，外层语法描述操作的控制结构，如顺序、选择、循环等，这些控制结构将加工中的各个操作连接起来。内层语法一般没有什么限制，就用自然语言描述。

(2) 判定表

有时"一幅图胜过千言万语"，判定表常用来描述一些不易用语言表达清楚或需要很大篇幅才能用语言表达清楚的加工。例如，在飞机票预订系统中，在旅游旺季的7—9，12月份，如果订票超过20张，优惠票价的15%；20张以下，优惠5%；在旅游淡季的1—6，10，11月份，订票超过20张，优惠30%；20张以下，优惠20%。见表3.2的判定表。

表 3.2

旅游时间	7—9，12月		1—6，10，11月	
订票量	≤20	>20	≤20	>22
折扣量	5%	15%	20%	30%

如表3.3所示，判定表分为四个区。Ⅰ区内列出所有的条件类别，Ⅱ区内列出所有的条件组合，Ⅲ区内列出所有的操作，Ⅳ区内列出在相应的组合条件下，某个操作是否执行或执行情况。在表3.2中，Ⅰ区的条件类别有两个：旅游时间和订票量，Ⅱ区内列出所有四种条件组合，Ⅲ区内只有一个操作，Ⅳ区标明在某种条件组合下操作的执行情况。

表 3.3

Ⅰ 条件类别	Ⅱ 条件组合
Ⅲ 操作	Ⅳ 操作执行

当描述的加工由一组操作组成，而且是否执行某些操作或操作的执行情况又取决于一组条件时，用判定表书写加工逻辑比较方便。表3.4是使用判定表的另一个例子。

表 3.4

考试总分	≥620	≥620	<620	<620
单科成绩	有满分	有不及格	有满分	有不及格
发升级通知书	Y	Y	N	N
发免修单科通知书	N	N	Y	N
发留级通知书	N	N	Y	Y
发重修单科通知书	N	Y	N	N

(3) 判定树

判定树用图形形式描述加工逻辑，其特点是结构清晰，易读易懂。判定表 3.2 可用图 3.3 的判定树等价表示。

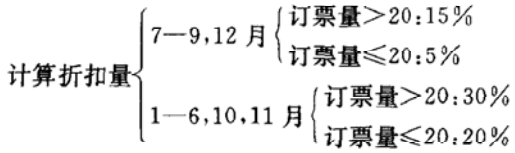


图 3.3 同判定表 3.2 等价的判定树表示

判定表 3.4 可用图 3.4 的判定树等价表示。

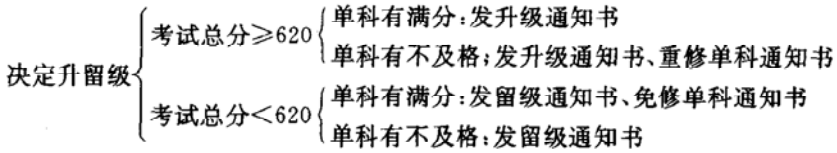


图 3.4 同判定表 3.4 等价的判定树表示

加工逻辑可以用语言、表格、图形等多种形式来描述，也可以将它们结合使用。在保证描述的小说明清晰易懂的前提下，可以自由选择描述方法。

3.2.2 实施步骤

结构化分析从本质上说是一种运用抽象和分解技术，“自顶向下、逐步求精”的过程。前面我们介绍了结构化分析的表示工具，有了工具以后，下面我们接着讨论如何进行结构化分析。

1. 确定系统边界，画出系统环境图

经过需求获取阶段的工作，分析员可以比较容易地定出系统边界，即系统与外界或环境的输入和输出，从而画出系统环境图，或称顶层数据流图。例如，上面的飞机票预订系统的顶层数据流图如图 3.5 所示。

图 3.5 一个飞机票预订系统的顶层数据流图

2. 自顶向下，画出各层数据流图

有了顶层数据流图后，下面的工作就是自顶向下画出各层的数据流图。具体地说，就是对

加工进行“逐层分解”，直到底层的加工足够简单，功能清晰易懂，不必再继续分解为止。

图 3.6 就是一个系统的分层数据流图。层次的编号是按顶层、0 层、1 层、2 层……的次序编排的。顶层图即系统环境图，标出了系统的边界；0 层图是顶层图中包含的唯一加工的细化，0 层图中的加工 2 又为 1 层图所细化。有时为方便起见，称这些图互为“父了”关系，即顶层图是 0 层图的“父图”，0 层图是 1 层包含的所有数据流图的“父图”；反过来，0 层图是顶层图的“子图”，1 层包含的所有数据流图是 0 层图的“子图”。除顶层图外，其它各层的数据流图都是某一父图的子图，这些数据流图统称为数据流子图或简称为子图。

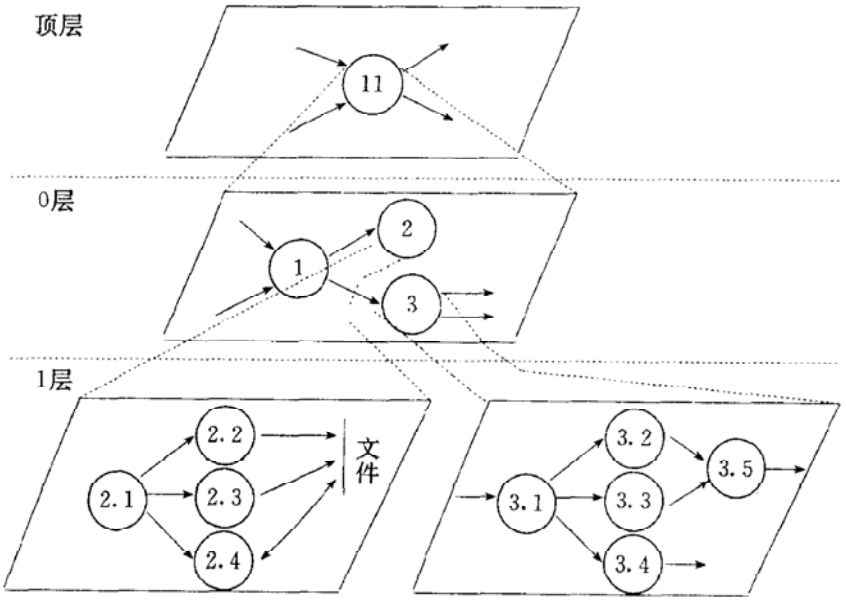


图 3.6　某个系统的分层数据流图

图 3.7 给出了一般系统的数据流图的总体结构：

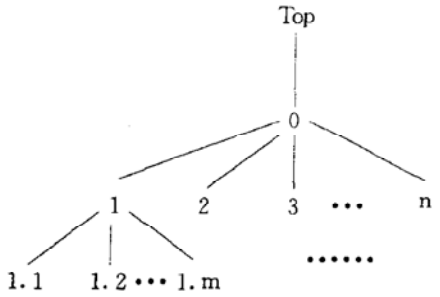


图 3.7　一般数据流图的总体结构

为了便于管理，需要给所有的数据流图和加工编号，并在整个系统中应是唯一的。本书是按下述规则为分层数据流图和图中的加工进行编号的：

① 顶层图不参与数据流图编号，顶层图中的唯一加工也不编号。

② 从 0 层图开始的所有子图和加工均需编号，子图的编号为分解的父图中相应加工的编号（如图 3.7 所示）。

③ 加工的编号由相应的子图号、小数点、加工在子图中的顺序号组成。

0层只有一张子图，按照规则，图中的加工编号依次为0.1，0.2，0.3…，为了方便通常去掉前面的0和小数点，则它们的编号依次为1，2，3…；1层子图的编号为0层细化的加工号，它们的编号就是1，2，3…，相应子图中的加工编号由了图号、小数点、加工在子图中的顺序号组成，依次类推。

3. 定义数据字典

按照前一节的系统模型，定义各层数据流图中包含的所有数据流和数据存储。

4. 定义小说明

按照前一节的系统模型，定义最底层数据流图中包含的所有加工。

5. 汇总前面各步骤的结果

以上我们讨论了建立系统模型的步骤，下面是一些在建立系统模型时的注意事项：

(1) 模型平衡规则

① 数据流图中所有的图形元素必须根据它们的用法规则正确使用。例如，一个加工必须既有输入又有输出；数据流只能和加工相关，即从加工流向加工、数据源流向加工或加工流向数据源。

② 每个数据流和数据存储都要在数据字典中有定义，数据字典将包括各层数据流图中数据元素的定义。

③ 数据字典中的定义使用合法的逻辑构造符号。

④ 数据流图中最底层的加工必须在小说明中有定义。

⑤ 父图和子图必须平衡，即父图中某加工的输入输出(数据流)和分解这个加工的子图的输入输出(数据流)必须完全一致，这种一致性不一定要求数据流的名称和个数一一对应，但它们在数据字典中的定义必须一致，数据流或数据项既不能多也不能少。

⑥ 小说明和数据流图的图形表示必须一致。例如，在小说明中，输入数据流必须说明其如何使用，输出数据流说明如何产生或选取，数据存储说明如何选取、使用或修改。

(2) 控制复杂性的一些规则

① 上层数据流可以打包(打包的数据流作一特殊标记)，上、下层数据流的对应关系用数据字典描述(编号对应)，同层的数据流也可编号对应，避免形成复杂的连线；只有一点限制，包内流的性质(输入、输出、输入输出)必须一致。

② 为了便于人的理解，把一幅图中的图元个数控制在7±2以内(Miller，1956)。

③ 检查同每个加工相关的数据流，并寻找是否有其它可降低界面复杂性的划分方法(有时一个加工有太多的输入输出数据流与同一层的其它加工或抽象层次有关)。

④ 分析数据内容，确定是否所有的输入信息都用于产生输出信息；相应地，由一个加工产生的所有信息是否都能由进入该加工的信息导出。

3.3 需求验证

需求分析阶段的工作结果是开发软件系统的重要基础，大量统计数字表明，软件系统中15%的错误起源于错误的需求。为了确保软件开发成功，提高质量和降低费用，一旦对目标系统提出一组需求之后，就必须严格验证这些需求的正确性。一般说来，应该从以下几个方面的

特征对软件需求规格说明书(Software Requirements Specification,以下简称SRS)加以验证。

1. 正确性

正确性指的是SRS中陈述的每个需求都表达了将要构造的系统的某种要求。目前尚不存在有效的的技术来保证这个质量,因为它完全依赖于当前的应用系统,例如,如果软件必须在5秒钟内对所有的按键事件作出响应,而SRS中陈述"软件应在10秒钟内对所有的按键事件作出响应",则该需求描述是不正确的。图3.8有助于以可视化的方式解释正确性的含义。左边的圆圈表示用户的实际需求,右边的圆圈表示SRS中陈述的需求。如果SRS是正确的,则区域C为空,即SRS中的每个需求都表达了未来系统的某种要求。

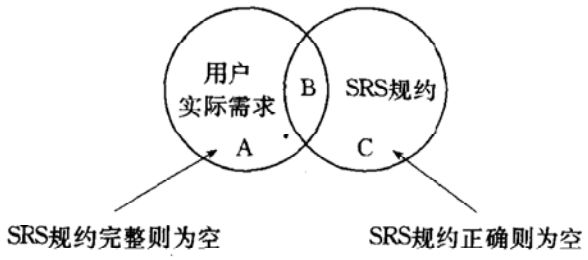


图3.8 用户实际需求——SRS规约

2. 无二义性

无二义性指的是SRS中陈述的每个需求都只有唯一的一种解释。设想从SRS中抽出一句话,把它交给10个人理解,如果存在一种以上的解释,那么这句话就可能是有二义性(或多义性)的。为克服SRS中具有的二义性,可以把一个SRS中所有具有二义性的术语统一收集在一起,并注明在不同情况下各个术语的准确含义。然而,同二义性有关的问题远非一个简单的术语汇编所能解决的。特别地,自然语言具有不可避免的二义性,因此使用自然语言书写SRS是容易导致二义性的。

3. 完整性

如果一个SRS具有以下三个特性,那么它是完整的:

① 期待未来系统所做的任何事情都包括在SRS的陈述中,完整性在图3.8中就意味着区域A为空。注意,如果一个SRS既是完整的又是正确的,那么区域A和C同时为空,图3.8中的两个圆是重合的。完整性是所有属性中最难以保证的,原因是不完整意味着有一些东西不在SRS中,而这很难通过检查现有的材料发现其中不存在的东西,唯一可能发现这种疏忽或遗漏的是那些需要软件解决他们的问题的用户。诊断不完整性的一个有效技术就是做原型。此外,原型还有助于对需求的其它特征进行验证。

② SRS中应包括未来软件系统在所有可能情况下对所有可能的输入的响应,所有可能的输入包括有效输入和无效输入。这就意味着对于SRS中提到的每个系统输入,都应说明合适的系统输出将会是什么。值得注意的是,系统输出可能不仅仅是输入的函数,还可能是当前系统状态的函数。例如,在一个电话转接系统中,当检测到用户拨号9时,系统响应是当前状态的函数,而系统的状态依次又是用户前面动作的函数,因此,如果用户电话机的话柄没有拿起,就没有系统输出产生(即,输入9被忽略了)。如果用户刚开始拨号,则系统输出可能是另外一种拨号音;如果用户已经开始拨一个电话号码,9被收集起来作为电话号码的一部分。换句话说,SRS必须建立从输入域(I)和状态域(S)的笛卡尔乘积到输出域(O)和状态域的笛卡尔乘积的

完整映射，即

$$SRS: I \times S \rightarrow O \times S$$

③ SRS 中没有任何内容被标为“待定”。只要可能就应尽量避免在 SRS 中插入“待定”这个词，当包含待定时，应同时附加上谁应负责对该内容的最后确定，以及在哪个日期以前完成。这种方法就保证了待定不被随意当做无限拖延完成 SRS 的借口，就像待定意味着“明天做”，而明天当然永远也不会来。通过在 SRS 中包含负有责任的人员的名字和完成日期，我们保证了待定会在将来某一时刻消失。

4. 可验证性

可验证性指的是 SRS 中陈述的每个需求都是可验证的。一个需求是可验证的，当且仅当存在一个有限代价的过程，人工或机器可以检查实际的软件产品是否符合用户的需求。首先，任何二义性必然导致不可验证性，例如，下面的陈述“产品应有易于使用的用户界面”是存在二义性的，因为“易于使用”的观点对于不同的人可以有很不相同的理解，所以最终也是不可验证的；第二，使用不可度量的量，如“通常”或“时常”，意味着不存在一个有限的测试过程，也就意味着是不可验证的，例如，下面的陈述“当按下按钮时，系统通常应亮起红灯”就是不可验证的，因为当你验证最终系统是否符合用户需求时，连续按下按钮 1000 次，如果红灯亮了 600 次，你可能就试图声明测试成功，然而，当你再按下按钮时，有可能红灯再也没有亮，换句话说，测试“通常”的唯一方法是按下按钮无数次；第三，任何等同于停机问题的需求是不能被验证的(Turing，1936)，例如，可以证明“程序将不会进入一个无限循环”等同于停机问题，因而是不可验证的。

5. 一致性

一致性指的是：① SRS 中陈述的需求没有同以前的文档发生冲突，如系统需求规格说明书；② SRS 中陈述的各个需求之间不发生冲突。

6. 可理解性

为了使一个 SRS 减少二义性、增强可验证性、完整性和一致性，我们应努力追求高度形式化的表示法。不幸的是，这样的表示法常常缺乏另一个重要的特性：非计算机专家对 SRS 的理解。在许多情况下，SRS 的主要读者是顾客或用户，他们通常是应用领域的专家而不是计算机专家。也许达到该目标的一个途径是使用形式化的表示法，并借助工具把形式化的 SRS 自动转换为等价的易于理解的内容，这就是 Balzer 及其合作者在 GIST 项目中采用的方法(1982)。如果在形式化表示和非形式化表示之间存在一个完整的无二义的映射，那么非形式化的表示将满足所有要求的属性，包括被非计算机专家所理解。

以上主要讨论了 SRS 内容方面的属性，下面集中讨论关于 SRS 格式和风格方面的属性。

7. 可修改性

可修改性指的是 SRS 的结构和风格使任何对需求的必要的修改都易于完整和一致地进行。可修改性意味着存在一个有关 SRS 内容的列表、索引和交叉引用，如果以后必须对一个需求进行修改的话，我们可以方便地检查并定位 SRS 中所有必须加以修改的部分。例如，假设我们想把电话转接系统中对用户拨号的最小响应时间从 5 秒改为 3 秒，我们将查看“用户拨号”下面的索引，定位文档中所有对“用户拨号”的引用，以做必要的修改。改进 SRS 可读性的一个技术是在文档中不同的地方重复特定的需求，SRS 的这个属性称为冗余。例如，在描述自动用户交换机的外部接口时，我们必须定义用户和电话机之间的交互，因此，当描述市内通话的外

部表现时，SRS 可能会作如下陈述：

用户首先找到一个空闲的电话机，拿起听筒，系统将发出拨号音，然后用户开始拨打受话方的 8 位电话号码……。

当描述长途通话的外部表现时，SRS 可能会作如下陈述：

用户首先找到一个空闲的电话机，拿起听筒，系统将发出拨号音，然后用户先拨 0，接着拨打受话方的 10 位电话号码……。

注意上面的描述，文档对拨打电话的前三个步骤的重复增加了可读性。然而，可读性的增加潜在地降低了可修改性，因为当以后只对其中某一处的改变将导致 SRS 的不一致。为使冗余成为可接受的，索引表或交叉引用表对多次出现的需求的定位是基本的。

8. 可被跟踪性

可被跟踪性指的是 SRS 中的每个需求的出处都是清楚的，这意味着 SRS 中包含对前期支持文档的引用表，如图 3.9 所示，让我们假设 SRS 中包含如下的需求陈述：

系统应对任何一次 X 请求在 20 秒内响应。

假设软件已被实际构造出来，当接受最终测试时，测得的响应时间是 60 秒，那么对这个问题有两种解决方案：①重新设计或编码软件，以使其效率更高，或，②把需求从 20 秒改为 60 秒。如果 SRS 在该需求陈述处没有提供任何引用，以表示 20 秒是从其它什么地方来的，而只是一个随意选定的时间约束，我们就可以选择第二种方法（并且该方法可能是一个相当满意的解决）。然而，如果该应用是一个实时病房监控系统，20 秒钟的响应时间是从前期的文档引用来的，在该实时病房监控系统安装的特定医院环境中，根据当前护士和病人的比例，每个护士每分钟至少需要对该系统查询 3 次，以保证任何病人的每次紧急情况都不会被遗漏。在这种情况下，可被跟踪性就要求在 SRS 中对时间约束的需求处插入对前期文档的引用，那么在考虑第二种解决方案时，当检查了前期的文档之后立即就可被否定。

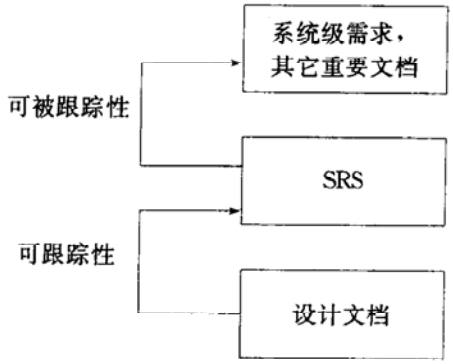


图 3.9　可跟踪性及可被跟踪性

9. 可跟踪性

为了设计或测试软件的任何成分，有必要清楚地知道该软件成分满足或部分满足哪个需求，同样地，当测试该软件系统时，有必要理解该测试针对哪些需求的有效性。可跟踪性指的是 SRS 的书写方式有助于对其中陈述的每个需求进行引用，如图 3.9 所示。存在许多达到该目标的技术和方法：

① 给每个段落按层次编号，以后当指明某个需求时，通过指出段落编号和句子在段落中

的序号即可，如需求 2.3.2.4S3 指的是在段落 2.3.2.4 中的第 3 个句子，这是一种非常有效的方法。

② 给每个段落按层次编号，在任何段落中都不包括多于一个需求，以后当指明某个需求时，只需给出段落编号即可。这种方法是可行的，但易于导致难以阅读的文档。

③ 在 SRS 中出现的每个需求都编以唯一的序号。

④ 使用一种指示需求的惯例表示法，即总是在包含一个需求的句子中使用“shall”，然后使用简单的 shall 提取工具，提取并唯一编码所有带 shall 的句子。

10. 设计无关性

设计无关性指的是 SRS 不暗示特定的软件结构或算法。每个需求都限制了设计的选择，例如，如果你想要一个在某建筑物中上下运送旅客的系统，就排除了电话转接系统软件作为一种可能的设计。SRS 中的任何需求都不应把设计限制在唯一的选择上，当然，SRS 的设计无关性也存在一些例外，特别是对于一些特殊的应用，构造一个新系统的主要动机是为了使用特定的体系结构或算法，例如，一个电梯控制系统的 SRS 必然包含调度算法；一个集成电路布局程序的 SRS 无疑会包括布局和布线算法。

11. 注释

注释向开发机构提供了每个需求是否重要的指导意见。有时一个软件开发机构会花费过多的时间以满足某个特定的需求，后来发现客户宁愿得到一个需求没有完全满足的按时交货的产品，而不愿 6 个月后得到一个需求完全满足的产品。如果每个需求的相对重要性一开始就确定了，那么上面的情况显然是可以避免的。这样做的一种方法是给 SRS 中的每个个别的需求标上 E(Essential，基本的)，D(Desirable，希望的)或 O(Optional，任选的)。

一个 SRS 达到以上的所有目标几乎是不可能的，例如，当我们试图排除不一致性和二义性时(通常通过减少 SRS 中的自然语言)，SRS 对于非计算机专家的用户将变得难以理解；当我们试图达到绝对的完整性时，SRS 和其它文档将变得十分庞大和难以阅读；如果我们通过排除冗余以增加可修改性，SRS 将变得难以琢磨和具有二义性。

我们可以得出的唯一结论是：不存在十全十美的 SRS！

3.4 需求分析文档

需求分析规格说明书是需求分析阶段产生的一份最重要的文档，它以一种一致的、无二义的方式准确地表达用户的需求。需求规格说明书主要起以下三方面的作用：

① 作为软件开发机构和用户之间一份事实上的技术合同书；

② 作为软件开发机构下一步进行设计和编码的基础；

③ 作为测试和验收目标系统的依据。

在本节中我们将给出一份典型的需求规格说明书的样例，供大家在实际工作中参考。

此外，在需求分析阶段一般还会产生另外两个文档——初步测试计划和用户系统描述。

在系统开发早期设计一个软件测试计划是十分必要的。大量的统计数字表明，在系统开发早期发现并修改一个错误的代价往往很低，越到系统开发的后期，改正同样错误所花费的代价越高，举个例子，假设在需求分析阶段检测并改正一个错误的代价为 1 个单位，那么到了软件测试阶段检测并改正同样的错误所花费的代价，据典型的保守数字为 10 个单位，而到软件发

布后的代价就可能高达100个单位。所以，尽可能地在系统开发的早期进行软件测试，就可以较小的代价检测出需求规格说明书中不可避免的错误。这个初步的测试计划应包括对未来系统中的哪些功能和性能指标进行测试，以及达到何种要求。在后阶段的软件开发中，对这个测试计划要不断地修正和完善，并成为相应阶段的文档的一部分。

用户系统描述从用户使用系统的角度描述系统，相当于一份初步的用户手册。内容包括对系统功能和性能的简要描述，使用系统的主要步骤和方法，以及系统用户的责任等等。在软件开发过程的早期，准备一份初步的用户手册是非常必要的，它使得未来的系统用户能够从使用的角度检查、审核目标系统，因此比较容易判断这个系统是否符合他们的需要。为了书写这份文档，也迫使系统分析员从用户的角度考虑软件系统。有了这份文档，审查和复审时就更容易发现不一致和误解的地方，这对保证软件质量和项目的成功是很重要的。

下面给出某系统需求规格说明书的实例。

××××××系统需求规格说明书

1. 引言
   - 1.1 编写目的

     说明编写本需求分析规格说明书的目的。
   - 1.2 背景说明

     (1) 给出待开发的软件产品的名称；

     (2) 说明本项目的提出者，开发者及用户；

     (3) 说明该软件产品将做什么，如有必要，说明不做什么。
   - 1.3 术语定义

     列出本文档中所用的专门术语的定义，和外文首字母组词的原词组。
   - 1.4 参考资料

     列出本文档中所引用的全部资料，包括标题、文档编号、版本号、出版日期及出版单位等，必要时注明资料来源。
2. 概述
   - 2.1 功能概述

     叙述待开发软件产品将完成的主要功能，并用方框图来表示各功能及其相互关系。
   - 2.2 约束

     叙述对系统设计产生影响的限制条件，并对下一节中所述的某些特殊需求提供理由，如管理模式、硬件限制、与其它应用的接口、安全保密的考虑等。
3. 数据流图与数据字典
   - 3.1 数据流图
     - 3.1.1 数据流图1
       - (1) 画出该数据流图
       - (2) 加工说明
         - (a) 编号
         - (b) 加工名
         - (c) 输入流
         - (d) 输出流

(e) 加工逻辑

(3) 数据流说明

3.1.2 数据流图 2

……

3.2 数据字典

3.2.1 文件说明

说明文件的成分及组织方式。

3.2.2 数据项说明

以表格的形式说明每一数据项，格式如下表所示：

名　称	类　型	含　义	度量单位	有效范围	精　度

4. 接口

4.1 用户接口

说明人机界面的需求，包括：

(1) 屏幕格式；

(2) 报表或菜单的页面打印格式及内容；

(3) 可用的功能键及鼠标。

4.2 硬件接口

说明该软件产品与硬件之间各接口的逻辑特点及运行该软件的硬件设备特征。

4.3 软件接口

说明该软件产品与其它软件之间接口，对于每个需要的软件产品，应提供：

(1) 名称

(2) 规格说明

(3) 版本号

5. 性能需求

5.1 精度

逐项说明对各项输入数据和输出数据达到的精度，包括传输中的精度要求。

5.2 时间特征

定量地说明本软件的时间特征，如响应时间、更新处理时间、数据传输、转换时间、计算时间等。

5.3 灵活性

说明本软件所具有的灵活性，即当用户需求(如对操作方式、运行环境、结果精度、时间特性等的要求)有某些变化时，本软件的适应能力。

6. 属性

6.1 可使用性

规定某些需求，如检查点、恢复方法和重启动性，以确保软件可使用。

6.2 保密性

规定保护软件的要素。

6.3 可维护性

规定确保软件是可维护的需求，如模块耦合矩阵。

6.4 可移植性

规定用户程序、用户接口的兼容方面的约束。

7. 其它需求

7.1 数据库

说明作为产品的一部分来开发的数据库的需求。如：

(1) 使用的频率；

(2) 访问的能力；

(3) 数据元素和文件描述；

(4) 数据元素、记录和文件的关系；

(5) 静态和动态组织；

(6) 数据保留要求。

7.2 操作

列出用户要求的正常及特殊的操作，如：

(1) 在用户组织中各种方式的操作；

(2) 后援和恢复操作。

7.3 故障及处理

列出可能发生的软件和硬件故障，并指出这些故障对各项性能指标所产生的影响及对故障的处理要求。

3.5 实例研究

1. 项目说明

图书管理系统旨在用计算机对图书进行管理，包括图书的购入、借阅、归还以及注销。管理人员可以查询某位读者、某种图书的借阅情况，还可以对当前图书借阅情况进行一些统计，给出统计表格，以便全面掌握图书的流通情况。鉴于篇幅所限，我们这里给出一个非常简化的图书管理系统的例子，旨在说明进行数据流分析的方法。

本系统针对图书主要进行四方面的管理：购入新书、读者借书、读者还书以及图书注销。①购入新书时需要为该书编制图书卡片，包括分类目录号、流水号（要保证每本书都有唯一的流水号，即使同类图书也是如此）、书名、作者、内容摘要、价格和购书日期等信息，写入图书目录文件中。②读者借书时填写借书单，包括读者号、欲借图书分类目录号，系统首先检查该读者号是否有效，若无效，则拒绝借书；否则进一步检查该读者所借图书是否超过最大限制数（此处我们假设每位读者同时只能借阅不超过五本书），若已达到最大限制数（此处为五本），则拒绝借书；否则读者可以借出该书，登记图书分类目录号、读者号和借阅日期等，写回到借书文件中去。③读者还书时，根据图书流水号，从借书文件中读出和该图书相关的借阅记录，表明还书日期，再写回借书文件中，如果图书逾期未还，则处以相应罚款。④在某些情况下，需要对图书馆的图书进行清理工作，对一些过时或无继续保留价值的图书要注销，这时可以从图书文件里删除相关记录。⑤咨询要求分为查询某位读者、某种图书和全局图书情况三种。

2. 数据流图

经过分析，得出的数据流图如图 3.10，3.11，3.12 和 3.13 所示。

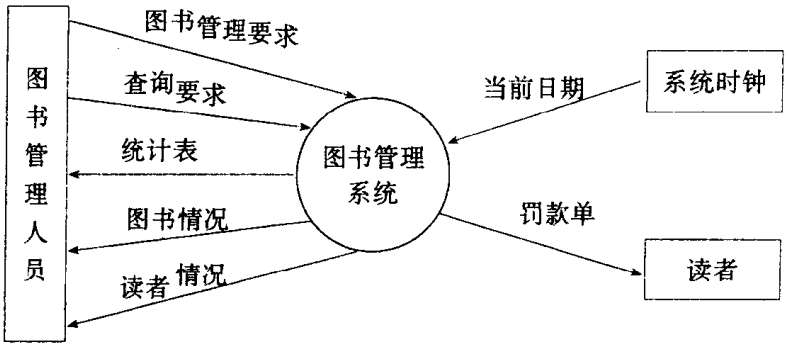


图 3.10　顶层数据流图

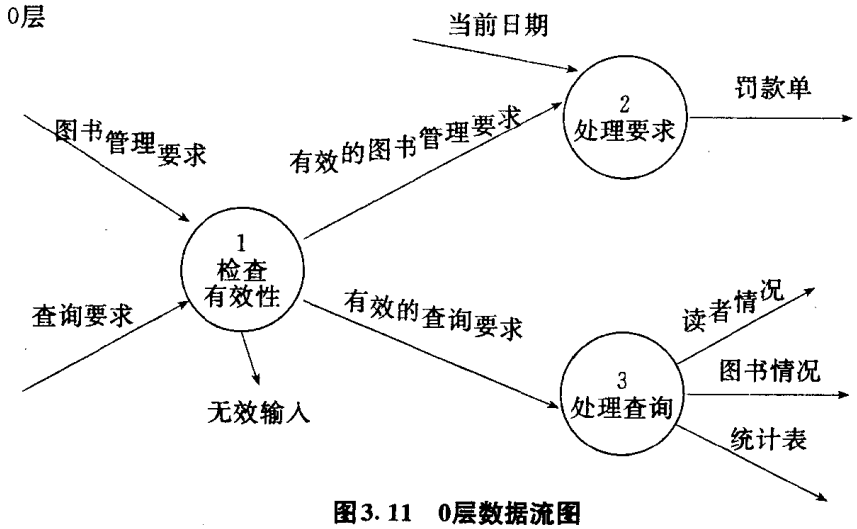


图3.11　0层数据流图

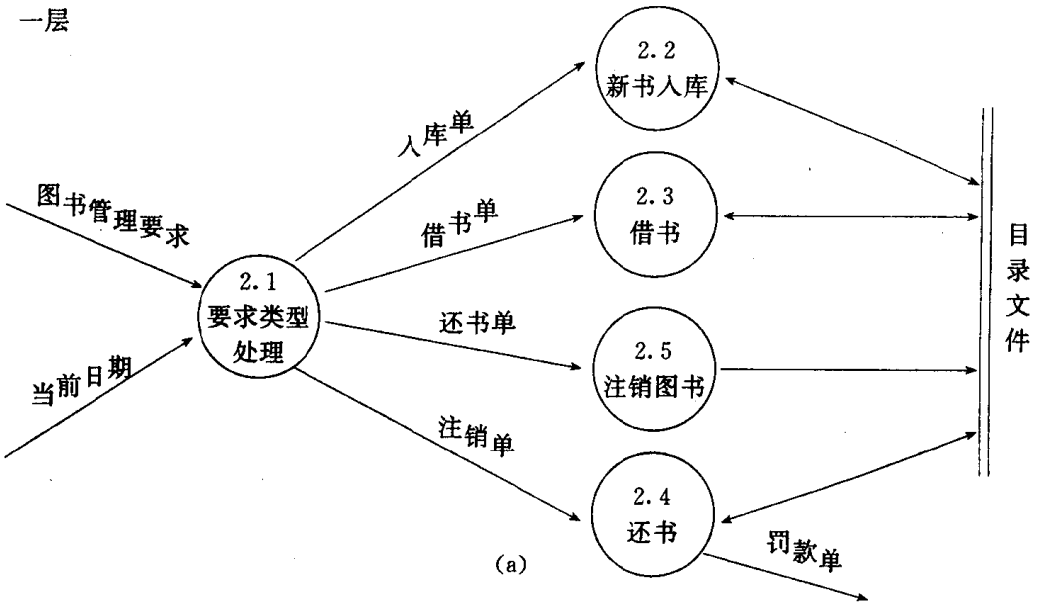


图3.12　一层数据流图

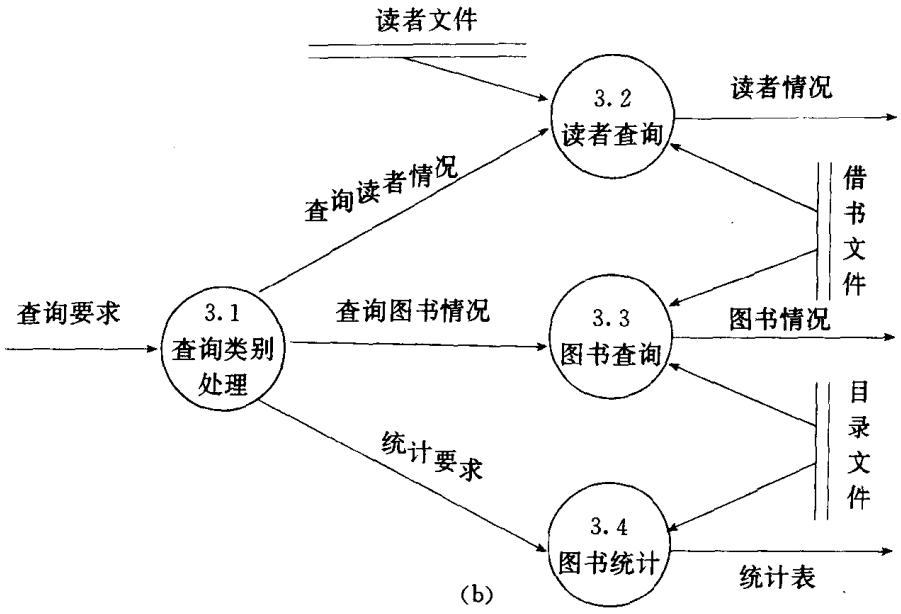


图3.12　一层数据流图

3. 数据字典

(1) 数据流条目

图书管理要求=〔入库单|借书单|还书单|注销单〕

入库单=分类目录号+数量+书名+作者+内容摘要+价格+购书日期

借书单=读者号+分类目录号+借阅日期

借书记录=读者号+分类目录号+图书流水号+借阅日期

还书单=图书流水号+还书日期

罚款单=逾期天数+罚款金额

注销单=图书流水号

查询要求=〔读者情况|图书情况|统计要求〕

读者情况=读者号+姓名+所在单位+{借书情况}

借书情况=书名+分类目录号+图书流水号+借阅日期

图书情况=书名+作者+分类目录号+总数+库存数

统计表={图书情况}

(2) 文件条目

文件名:读者文件

组成:{读者号+姓名+所在单位}

组织:按读者号递增顺序排列

文件名:目录文件

组成:{分类目录号+书名+作者+内容摘要+价格+入库日期+总数+库存数+{图书流水号}}

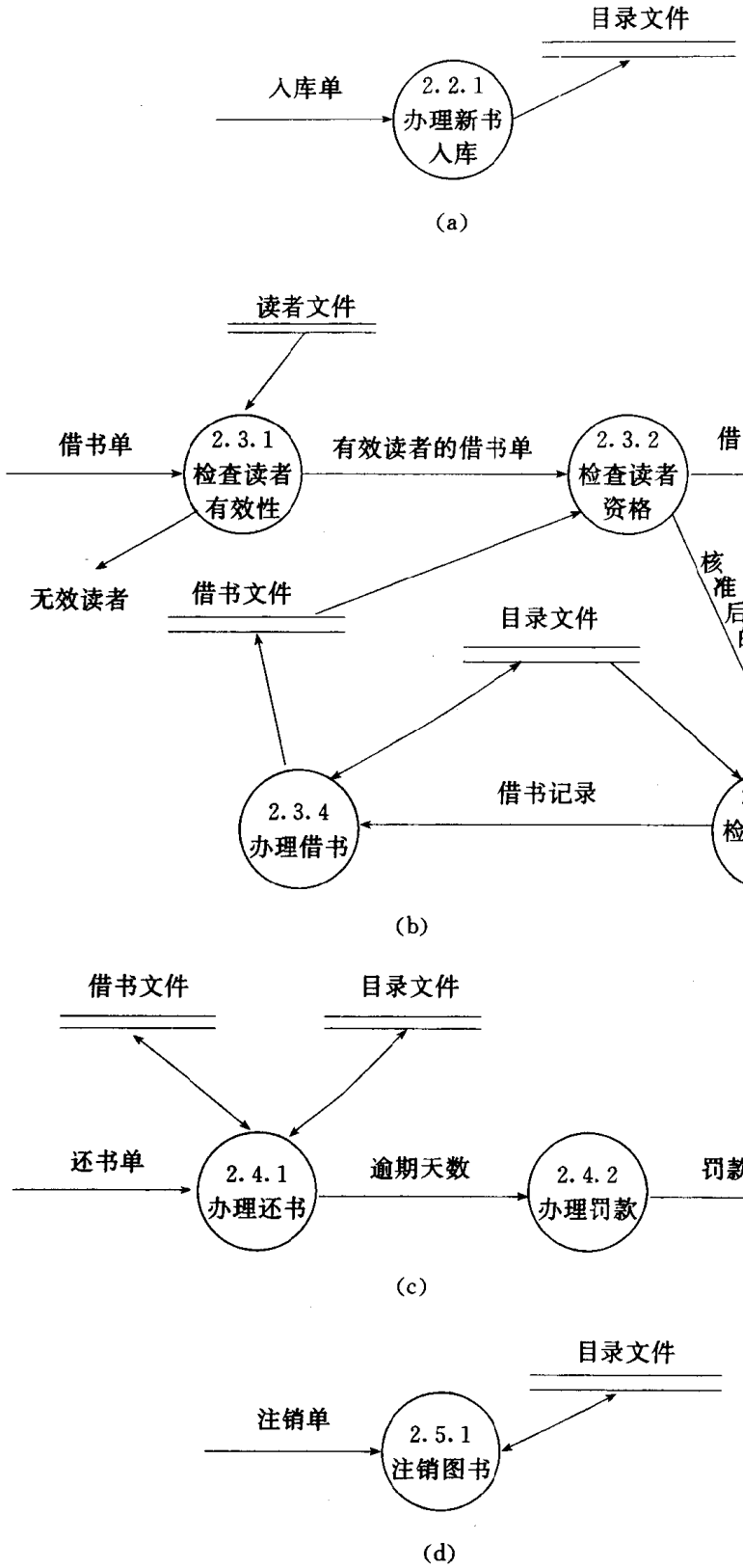

目录文件
入库单
2.2.1
办理新书
入库
(a)
读者文件
借书单
2.3.1
检查读者
有效性
有效读者的借书单
2.3.2
检查读者
资格
借书超过最大限制
无效读者
借书文件
目录文件
核
准
后
的
借
书
单
2.3.4
办理借书
借书记录
2.3.3
检查图书
库存
该书已借完
(b)
借书文件
目录文件
还书单
2.4.1
办理还书
逾期天数
2.4.2
办理罚款
罚款单
(c)
目录文件
注销单
2.5.1
注销图书
(d)

图 3.13　二层数据流图

组织：按分类目录号的字母顺序排列

文件名：借阅文件
组成：{借书记录＋还书日期}
组织：按借阅日期顺序排列

4. 小说明

小说明只描述最底层的基本加工。

加工编号：1
加工名：检查有效性
输入流：图书管理要求，查询要求
输出流：有效的图书管理要求，有效的查询要求
加工逻辑：检查输入要求的有效性

加工编号：2.1
加工名：要求类型处理
输入流：图书管理要求，当前日期
输出流：入库单，借书单，还书单，注销单
加工逻辑：根据图书管理要求的类型选择
    case 1：新书入库，输出入库单
    case 2：借书，输出借书单
    case 3：还书，输出还书单
    case 4：注销图书，输出注销单

加工编号：3.1
加工名：查询类别处理
输入流：查询要求
输出流：查询读者情况，查询图书情况，统计要求
加工逻辑：根据查询类别选择
    case 1：查询读者情况
    case 2：查询图书情况
    case 3：统计要求

加工编号：3.2
加工名：读者查询
输入流：查询读者情况，读者文件，借书文件
输出流：读者情况
加工逻辑：根据查询读者的情况从读者文件中读出读者记录，并从借书文件中读出该读者的借书记录，综合输出该读者的借阅情况

加工编号：3.3
加工名：图书查询
输入流：查询图书情况，借书文件，目录文件

输出流：图书情况

加工逻辑：根据查询图书的情况从目录文件中读出该书信息，并从借书文件中读出该书的借阅记录，综合输出该书的借阅情况

加工编号：3.4

加工名：图书统计

输入流：统计要求，目录文件

输出流：统计表

加工逻辑：根据统计要求从目录文件中读出所有图书的记录，输出统计表

加工编号：2.2.1

加工名：办理新书入库

输入流：入库单

输出流：目录文件

加工逻辑：输入填写好的入库单，并写入目录文件

加工编号：2.3.1

加工名：检查读者有效性

输入流：借书单，读者文件

输出流：有效读者的借书单

加工逻辑：根据借书单上的读者号和读者文件的内容，检查该读者是否为合法读者

加工编号：2.3.2

加工名：检查读者资格

输入流：有效读者的借书单，借书文件

输出流：核准后的借书单

加工逻辑：从借书文件中读出该读者的当前借书情况，检查他所借图书是否已超过最大限制

加工编号：2.3.3

加工名：检查图书库存

输入流：核准后的借书单，目录文件

输出流：借书记录

加工逻辑：根据借书单上的分类目录号和目录文件的内容，检查该书是否还有库存；若有，则填写借书记录

加工编号：2.3.4

加工名：办理借书

输入流：借书记录，目录文件

输出流：目录文件，借书文件

加工逻辑：根据借书记录的内容，对目录文件中该书的库存数量减1，同时写入借书文件

加工编号：2.4.1

加工名：办理还书

输入流：还书单，借书文件，目录文件

输出流：借书文件，目录文件，逾期天数

加工逻辑：根据还书单，对目录文件中该书的库存数量加1，同时把借书文件中相应记录置为无效；根据借阅日期和当前日期计算该图书是否已过期，并输出逾期天数

加工编号：2.4.2

加工名：办理罚款

输入流：逾期天数

输出流：罚款单

加工逻辑：根据图书过期天数，开具罚款单

加工编号：2.5.1

加工名：注销图书

输入流：注销单，目录文件

输出流：目录文件

加工逻辑：根据图书注销单，从目录文件中删除相应记录

第四章　总 体 设 计

需求分析阶段的主要任务是确定系统必须“做什么”，形成软件的需求规格说明书。软件设计阶段的主要任务是确定系统“怎么做”，从软件需求规格说明书出发，形成软件的具体设计方案。软件设计可以采用多种方法，如结构化设计方法、面向数据结构的设计方法、面向对象的设计方法等，本章我们主要讨论结构化设计方法。

结构化软件设计可以进一步分为总体设计和详细设计两个阶段，总体设计确定系统的整体模块结构，但这时每个模块仍然处于“黑盒子”级，描述这些黑盒子里的具体内容是详细设计阶段的任务。本章我们主要讨论软件的总体设计。

4.1　总体设计的任务

总体设计阶段的主要任务是把系统的功能需求分配给软件结构，形成软件的模块结构图，如图 4.1 所示。在结构图中，矩形表示功能单元，称为“模块”；连接上下层模块的线段表示它们之间的调用关系。处于较高层次的是控制(或管理)模块，它们的功能相对复杂而且抽象；处于较低层次的是从属模块，它们的功能相对简单而且具体。依据控制模块的内部逻辑，一个控制模块可以调用一个或多个下属模块；同时，一个下属模块也可以被多个控制模块所调用，即尽可能地复用已经设计出的低层模块。

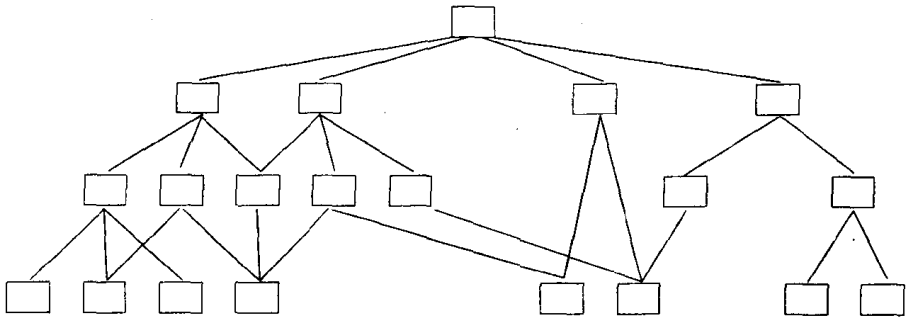


图 4.1　软件的模块结构图

在总体设计阶段，每个模块还处于黑盒子级，模块通过外部特征标识：名字，输入和输出。使用黑盒子的概念，设计人员可以站在较高的层次上进行思维，从而避免过早地陷入具体的条件逻辑、算法和过程步等实现细节，能够更好地确定模块和模块间的结构。

4.2　总体设计的表示形式

本节介绍在总体设计阶段可能会使用的几种表示形式。

4.2.1 层次图

层次图是在软件总体设计阶段最常使用的表示形式之一，用来描绘软件的层次结构。图中的每个方框代表一个模块，方框间的连线表示模块的调用关系。图 4.2 是层次图的一个例子，最顶层的方框代表正文加工系统的主控模块，它调用下层模块完成正文加工的全部功能；第二层的每个模块控制完成正文加工的一个主要功能，例如“编辑”模块通过调用它的下属模块可以完成六种编辑功能中的任何一种。

层次图很适合于在自顶向下设计软件的过程中使用。

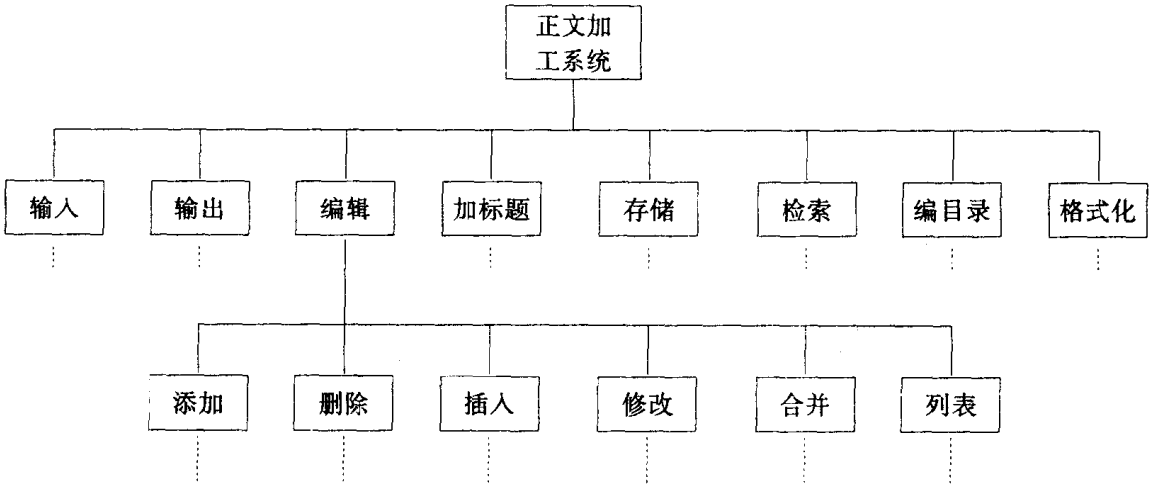


图 4.2 正文加工系统的层次图

4.2.2 HIPO 图

HIPO 图是由美国 IBM 公司发明的“层次图＋输入/处理/输出图”的英文缩写，从名字可以看出，HIPO 图实际上由 H 图和 IPO 图两部分组成，H 图就是上一节讲的层次图。为了能使 HIPO 图具有可跟踪性，在 H 图(层次图)里除了最顶层的方框之外，每个方框都加了编号。编号规则如下：最顶层方框不编号，第一层中各模块的编号依次为 1.0，2.0，3.0…；如果模块 2.0 还有下层模块，那么下层模块的编号依次为 2.1，2.2，2.3…；如果模块 2.2 又有下层模块，那么下层模块的编号依次为 2.2.1，2.2.2，2.2.3…，依此类推，如图 4.3 所示。

和 H 图中的每个方框相对应，应该有一张 IPO 图描述这个方框代表的模块的处理过程。IPO 图使用的符号既少又简单，能够方便地描述数据输入、数据处理和数据输出之间的关系。它的基本形式是：在左边的框中列出有关的输入数据，在中间的框中列出主要的处理——处理框中列出的处理次序暗示了执行的次序，在右边的框中列出产生的输出数据。另外，还用类似向量符号的粗大箭头清楚地指出数据通信的情况。图 4.4 是一个主文件更新的例子，通过这个例子不难了解 IPO 图的用法。

值得强调的是，HIPO 图中的每张 IPO 图内都应该明显地标出它所描绘的模块在 H 图中的编号，以便跟踪了解这个模块在软件结构中的位置。

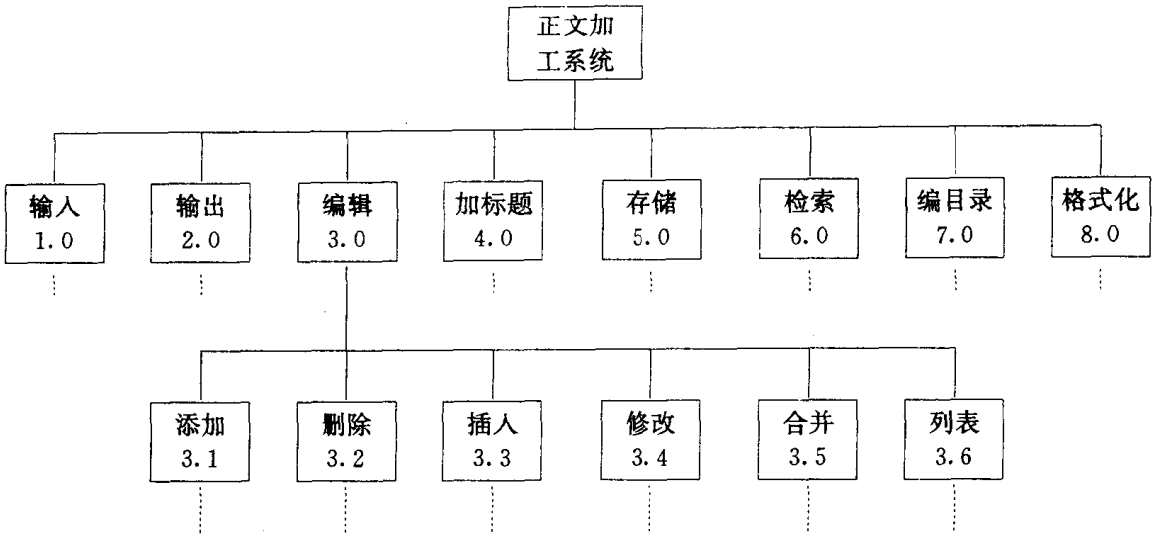


图 4.3　带编号的层次图(H 图)

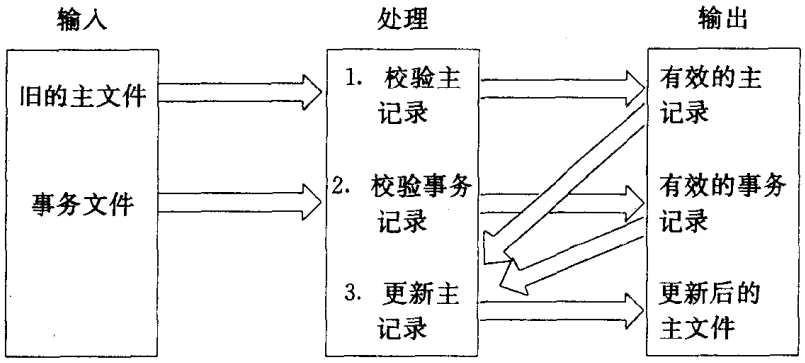


图 4.4　IPO 图的一个例子

4.2.3　结构图

Yourdon 提出的结构图是进行软件结构设计的另一个有力的表示方式。结构图和层次图类似，也是描述软件结构的图形表示，但描述能力比层次图更强。图中每个方框代表一个模块，框内注明模块的名字或主要功能；方框之间的箭头(或直线)表示模块的调用关系。因为按照惯例总是图中位于上方的方框代表的模块调用下方的模块，即使不用箭头也不会产生二义性，为了简单起见，可以只用直线而不用箭头表示模块之间的调用关系。

在结构图中通常还用带注释的箭头表示模块调用过程中来回传递的信息，如果希望进一步标明传递的信息是数据还是控制信息，则可以利用注释箭头尾部的形状来区分：尾部是空心圆表示传递的是数据，实心圆表示传递的是控制信息。图 4.5 是结构图的一个例子。

以上介绍的是结构图的基本符号，也是最经常使用的符号。此外还有一些附加的符号，可以表示模块的选择调用或循环调用。图 4.6 表示当模块 M 中某个判定为真时调用模块 A，为假时调用模块 B，图 4.7 表示模块 M 循环调用模块 A，B 和 C。

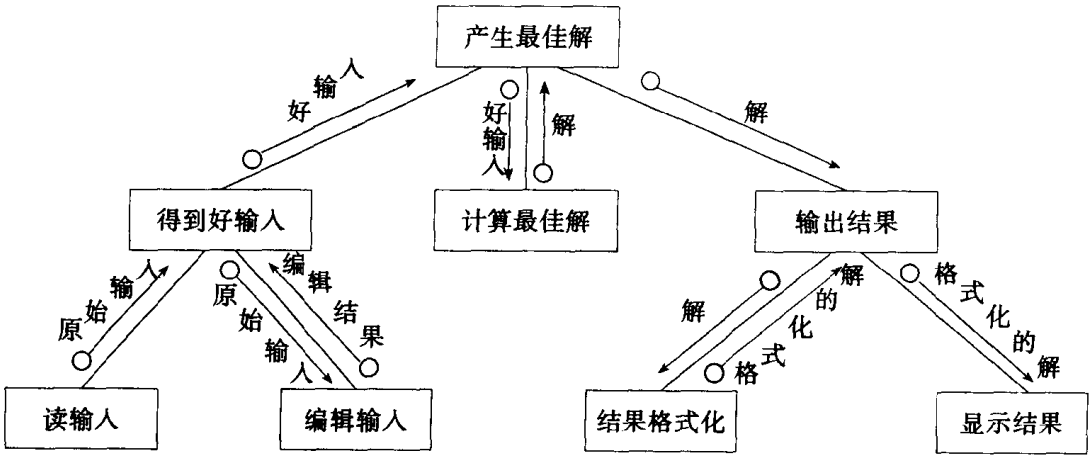


图4.5 结构图的例子——产生最佳解的一般结构

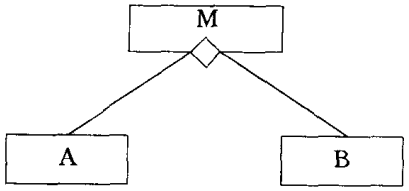


图4.6 判定为真时调用A，为假时调用B

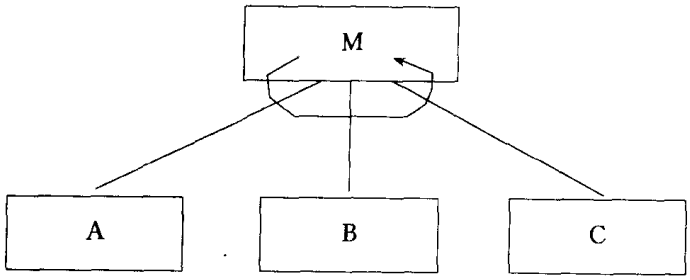


图4.7 模块M循环调用模块A，B，C

注意，层次图和结构图并不严格地表示模块的调用次序。虽然多数人习惯于按调用次序从左到右画模块，但并没有这种规定，出于其它方面的考虑(例如为了减少交叉线)，也完全可以不按这种次序画。此外，层次图和结构图并不指明什么时候调用下层模块。通常上层模块中除了调用下层模块的语句外还有其它语句，究竟是先执行调用下层模块的语句还是先执行其它语句，在图中丝毫没有指明。事实上，层次图和结构图只表明一个模块调用哪些模块，至于模块内还有没有其它成分则完全没有表示出来。

4.3 总体设计的方法

在需求分析阶段，信息流是一个关键的考虑，通常用数据流描述信息在系统中加工之间的

流动情况。结构化设计方法定义了一些不同的“映射”方法，利用这些映射方法可以把数据流图变换成软件结构，所以这种方法有时也称为面向数据流的设计方法。

4.3.1 数据流图的类型

面向数据流的设计方法把数据流图映射成软件结构，数据流图的类型决定了映射的方法。数据流图可以分下述两种类型：

1. 变换型数据流图

具有较明显的输入、变换（或称主加工）和输出界面的数据流图称为变换型数据流图。如图4.8所示，信息沿输入通路进入系统，同时由外部形式变换成内部形式，进入系统的信息通过变换中心，经加工处理以后再沿着输出通路变换成外部形式离开软件系统。

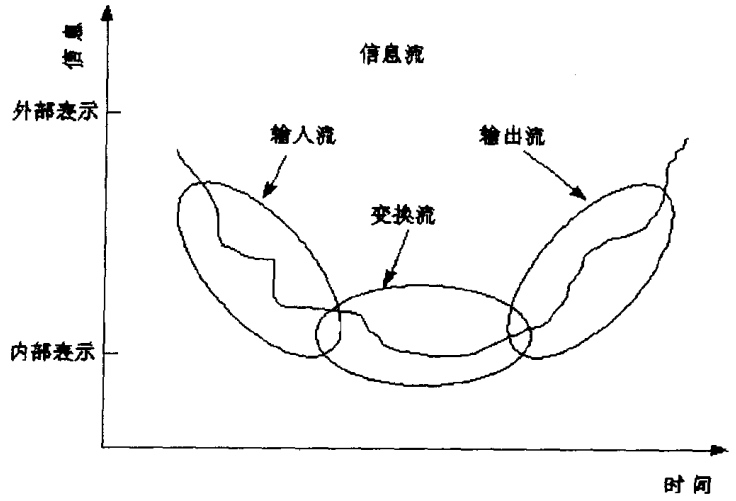


图4.8 交换流

2. 事务型数据流图

任何软件系统从本质上来说都是信息的变换装置，因此，原则上所有的数据流图都可以归为变换型。但是，当数据流图具有和图4.9类似的形状时，即数据沿输入通路到达一个处理T，这个处理根据输入数据的类型在若干动作序列中选出一个来执行，这类数据流图应该划为一类特殊的数据流图，称为事务型数据流图。图4.9中的处理T称为事务中心，它完成下述任务：

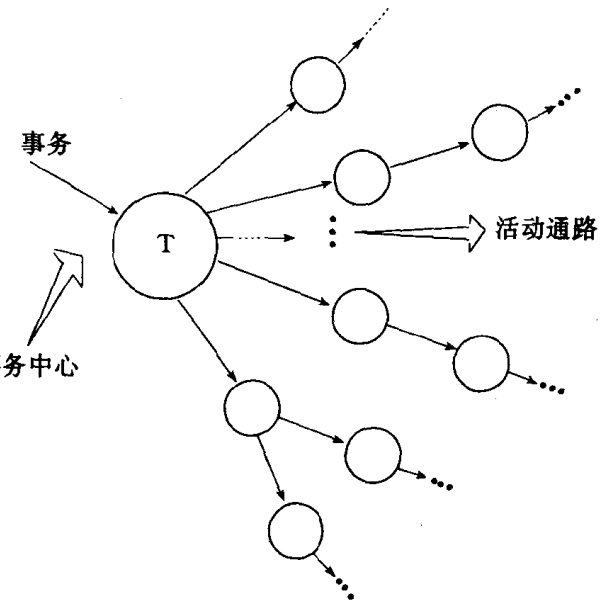


图4.9 事务流

- 接收输入数据（输入数据又称为事务）；
- 分析每个事务以确定它的类型；
- 根据事务类型选取一条活动通路。

4.3.2 设计步骤

图 4.10 说明了结构化设计方法的设计步骤。值得注意的是，任何设计过程都不是机械地一成不变的，设计首先需要设计人员的经验、判断力和创造精神，这往往是凌驾于方法的规则之上的。

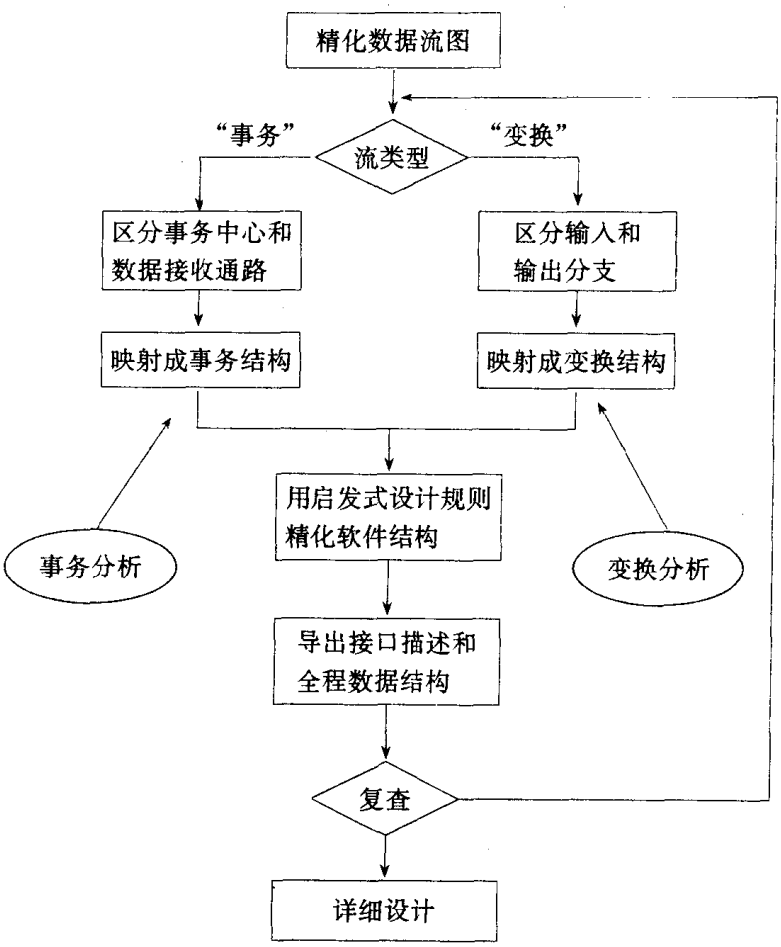


图4.10 面向数据流方法的设计过程

1. 变换设计

变换设计是一系列设计步骤的总称，经过这些步骤把变换型数据流图按预先确定的模式映射成软件结构。下面通过一个例子说明变换设计的方法。

(1) 例子

我们已经开始进入"智能化"产品时代，在这类产品中把软件做在只读存储器中，成为设备的一部分，从而使设备具有某些"智能"。因此，这类产品的设计都包含软件开发的任务。作为面向数据流的设计方法中变换设计的例子，考虑汽车数字仪表板的设计。

假设的仪表板将完成下述功能：

① 通过模-数转换实现传感器和微处理机的接口；

② 在发光二极管面板上显示数据；

③ 指示每小时英里数(mph)，行驶的里程，每加仑油行驶的英里数(mpg)等等；

④ 指示加速或减速；

⑤ 超速警告：如果车速超过 55 公里/小时，则发出超速警告铃声。

在软件需求分析阶段应该对上述每条要求以及系统的其它特点进行全面的分析评价，建立起必要的文档资料，特别是数据流图。

（2）设计步骤

第1步　复查基本系统模型。

复查的目的是确保系统的输入数据和输出数据符合实际。

第2步　复查并精化数据流图。

应该对需求分析阶段得出的数据流图认真复查，并且在必要时进行精化。不仅要确保数据流图给出目标系统的正确的逻辑模型，而且应该使数据流图中每个处理都代表一个规模适中相对独立的子功能。

假设在需求分析阶段产生的数字仪表板系统的数据流图如图4.11所示。

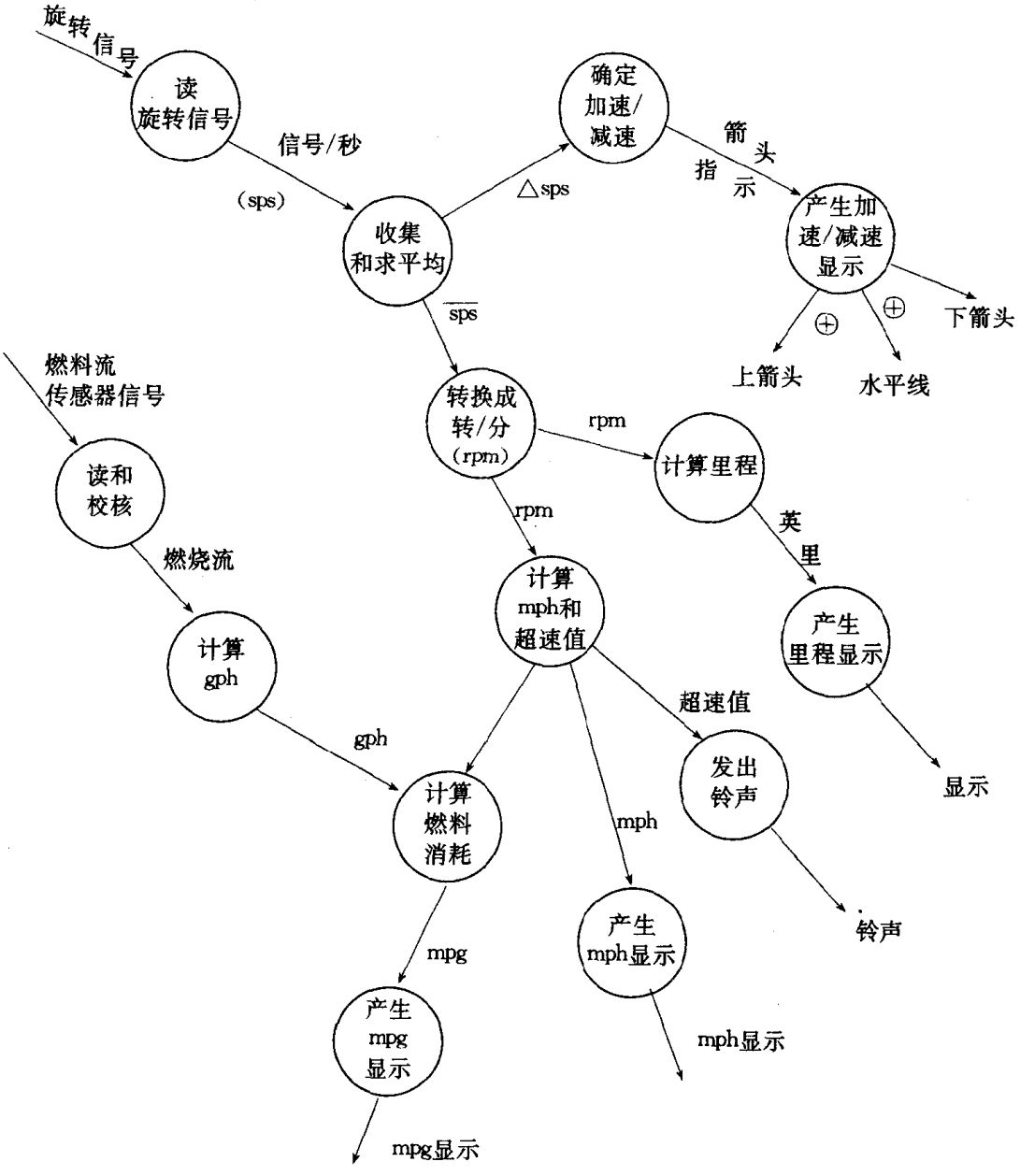


图4.11　数字仪表板系统的数据流图

这个数据流图对于软件结构设计的“第一次分割”而言已经足够详细了，因此不需要精化就可以进行下一个设计步骤。

第3步　确定数据流图具有变换特性还是事务特性。

一般地说，一个系统中的所有信息流都可以认为是变换流，但是，当遇到有明显事务特征的信息流时，建议采用事务分析方法进行设计。在这一步，设计人员应该根据数据流图中占优势的属性，确定数据流的全局特性。此外还应该把具有和全局特性不同特点的局部区域孤立出来，以后可以按照这些子数据流的特点精化根据全局特性得出的软件结构。

从图4.11可以看出，数据沿着两条输入通路进入系统，然后沿着五条通路离开，没有明显的事务中心。因此可以认为这个信息流具有变换流的总特征。

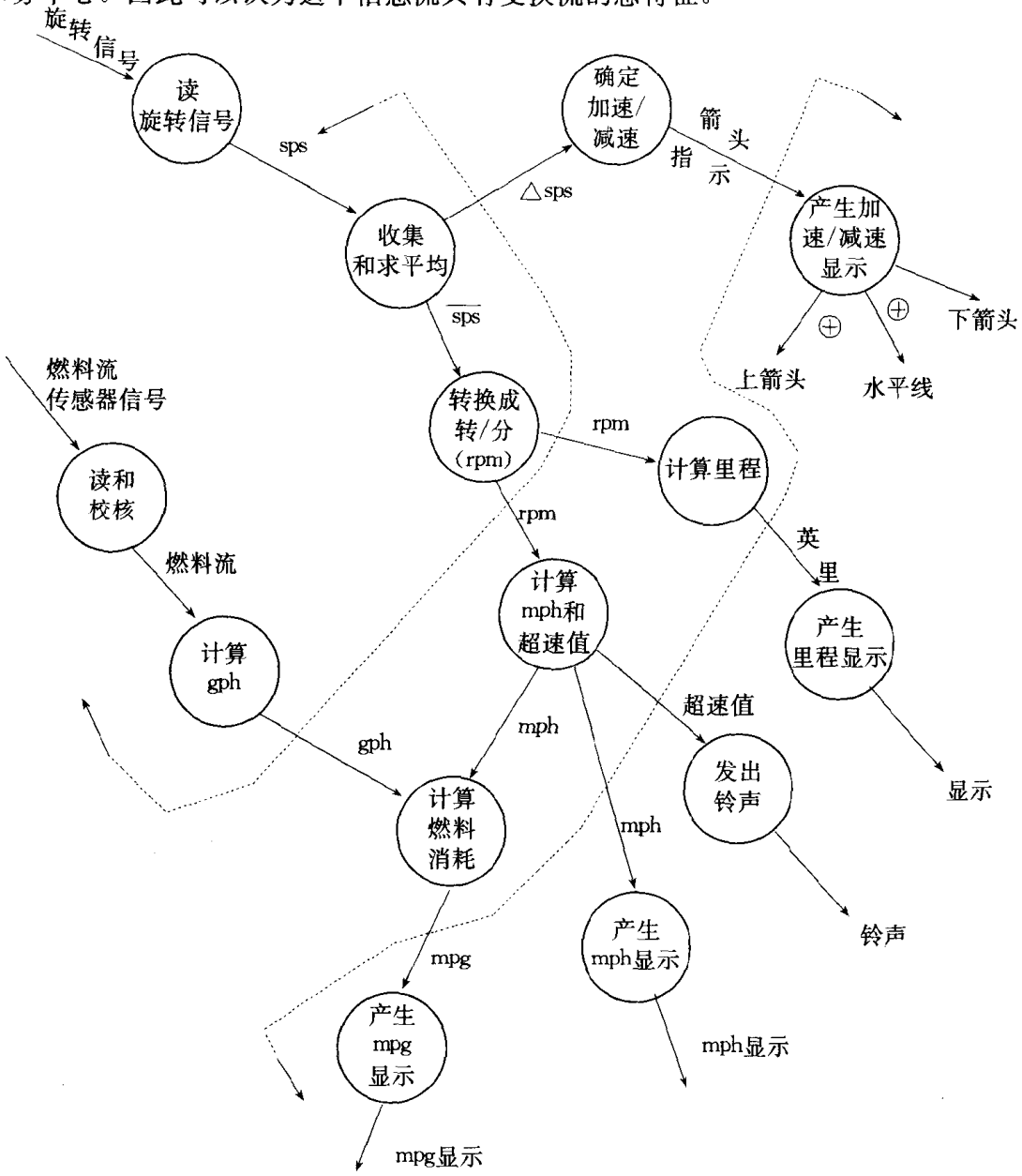


图4.12　具有边界的数据流图

第4步　确定输入流和输出流的边界，从而孤立出变换中心。

输入流和输出流的边界和对它们的解释有关，也就是说，不同的设计人员可能会在流内选取稍微不同的点作为边界的位置。当然在确定边界时应该仔细认真，但是把边界沿着数据流通路移动一个处理框的距离，通常对最后的软件结构只有很小的影响。

对于汽车数字仪表板的例子，设计人员确定的流的边界如图 4.12 所示。

第 5 步　完成“第一级分解”。

软件结构代表对控制的自顶向下的分配，所谓分解就是分配控制的过程。

对于变换流的情况，数据流图被映射成一种特殊的软件结构，这种结构控制输入、变换和输出等信息处理过程。图 4.13 说明了第一级分解的方法。

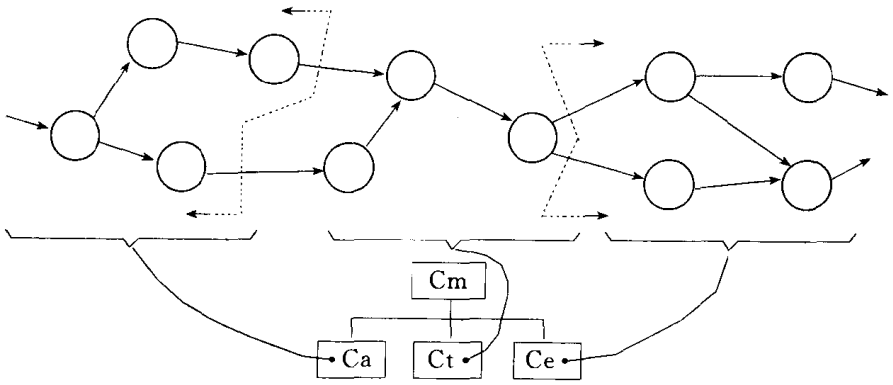


图 4.13　第一级分解的方法

- 位于软件结构最顶层的控制模块 Cm 协调下述从属模块的控制功能；
- 输入信息处理控制模块 Ca，协调对所有输入数据的接收；
- 变换中心控制模块 Ct，管理对内部形式的数据的所有操作；
- 输出信息处理控制模块 Ce，协调输出信息的产生过程。

虽然图 4.13 意味着一个三叉的控制结构，但是，对一个大型系统中的复杂数据流可以用两个或多个模块完成上述一个模块的控制功能。

对于数字仪表板的例子，第一级分解得出的结构如图 4.14 所示。每个控制模块的名字表明了为它所控制的那些模块的功能。

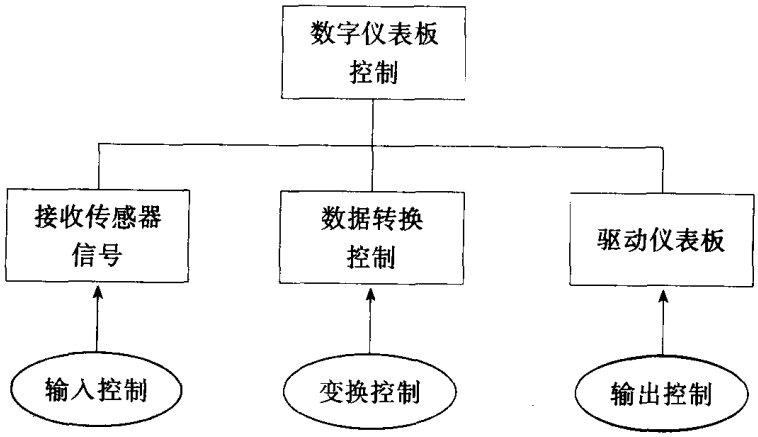


图 4.14　数字仪表板系统的第一级分解

第6步　完成“第二级分解”。

所谓第二级分解就是把数据流图中的每个处理映射成软件结构中一个适当的模块。完成第二级分解的方法是，从变换中心的边界开始沿着输入通路向外移动，把输入通路中每个处理映射成软件结构中 Ca 控制下的一个低层模块；然后沿输出通路向外移动，把输出通路中每个处理映射成直接或间接受模块 Ce 控制的一个低层模块；最后把变换中心内的每个处理映射成受 Ct 控制的一个模块。图 4.15 表示进行第二级分解的普遍途径。

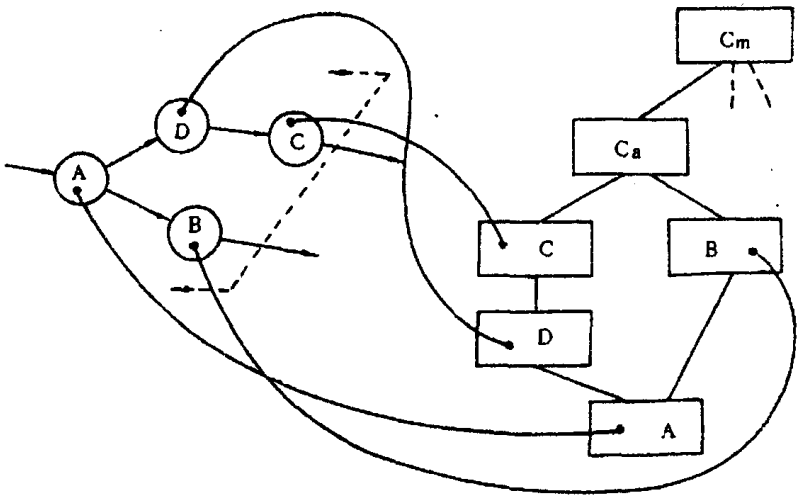


图 4.15　第二级分解的方法

对于数字仪表板系统的例子，第二级分解的结果分别用图 4.16，4.17 和 4.18 描述。这三张图表示对软件结构的初步设计结果。虽然图中每个模块的名字表明了它的基本功能，但是仍然应该为每个模块写一个简要说明，描述：

- 进入该模块的信息（接口描述）；
- 模块内部的信息；
- 过程陈述，包括主要判定点及任务等；
- 对约束和特殊点的简短讨论。

这些描述是初步的设计规格说明，在设计时期进一步的精化和补充是经常发生的。

第7步　使用设计度量和启发式规则对前面分解得到的软件结果进一步精化（下面第 4.6 节我们会详细讨论）。

上述七个设计步骤的目的是，开发出软件的整体表示。也就是说，一旦确定了软件结构就可以把它作为一个整体来复查，从而能够评价和精化软件结构。在这个时期进行修改只需要很少的附加工作，但是却能够对软件的质量特别是软件的可维护性产生深远的影响。

至此读者应该暂停片刻思考上述设计途径和“写程序”的差别。如果程序代码是对软件的唯一描述，那么软件开发人员将很难站在全局的高度来评价和精化软件，而且事实上也不能做到“既见树木又见森林”。

2. 事务设计

虽然在任何情况下都可以使用变换设计的方法设计软件结构，但是在数据流具有明显的事务流特点时，也就是有一个明显的“发射中心”（事务中心）时，最好采用事务设计的方法。

事务设计的步骤和变换设计的步骤大体相同，主要区别仅在于由数据流图映射到软件结

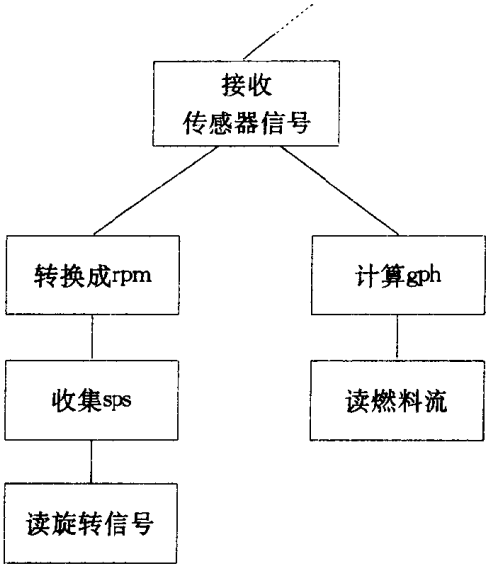


图4.16　未经细化的输入结构

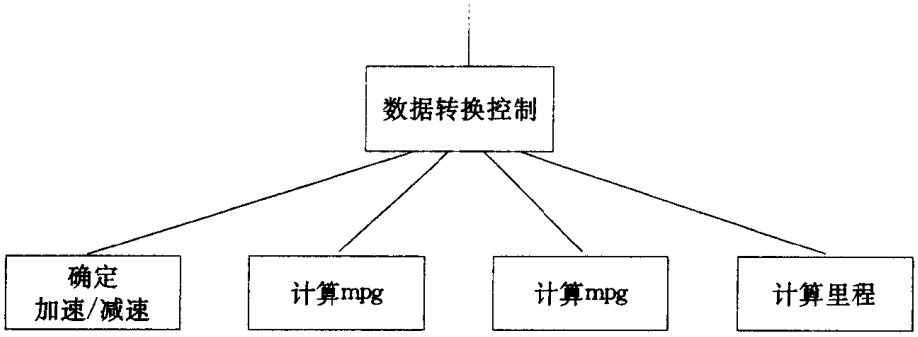


图4.17　未经细化的变换结构

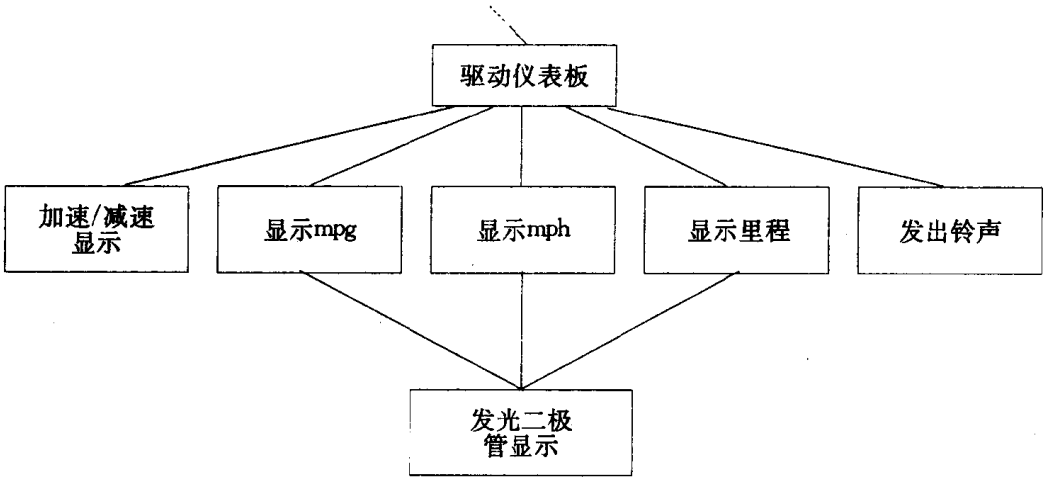


图4.18　未经细化的输出结构

构的方法不同。

由事务流映射成的软件结构包括一个接收分支和一个发送分支。映射出接收分支结构的

方法和变换设计映射出输入结构的方法很相像，即从事务中心的边界开始，把沿着接收流通路的处理映射成模块。发送分支的结构包含一个调度模块，它控制下层的所有活动模块；然后把数据流图中的每个活动流通路映射成与它的流特征相对应的结构。图 4.19 说明了上述映射过程。

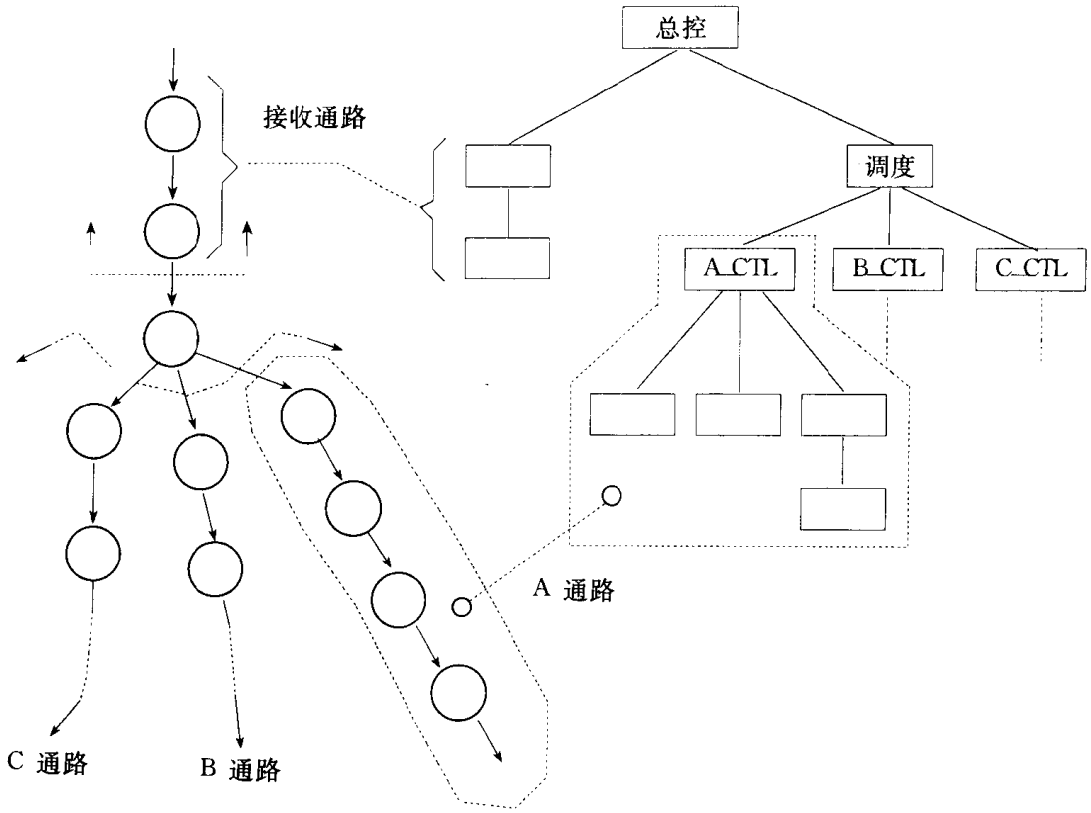


图4.19　事务设计的映射方法

对于一个大的系统，常常把变换设计和事务设计应用到同一个数据流图的不同部分，由此得到的子结构形成“构件”，可以利用它们构造完整的软件结构。

一般来说，如果数据流不具有显著的事务特点，最好使用变换设计；反之，如果具有明显的事务中心，则应该采用事务设计技术。但是，机械地遵循变换设计或事务设计的映射规则，很可能会得到一些不必要的控制模块，如果它们确实用处不大，那么可以而且应该把它们合并。反之，如果一个控制模块的功能过分复杂，则应该分解为两个或多个控制模块，或者增加中间层次的控制模块。

4.4　好的设计的准则

结构化设计的一个重要组成部分是它提供了衡量软件设计质量的广泛的评价准则，而这正是许多其它的软件设计方法所欠缺的，这也是结构化设计方法获得广泛应用的一个重要原因。评价软件设计质量的主要准则包括：

1. 模块化(Modularity)

模块化是好的软件设计的一个基本准则。高层模块使我们能从整体上把握问题，隐蔽细节以免分散我们的注意力；当我们需要时，又可以深入到较低的层次以了解进一步的细节。模块化提供了我们所需要的灵活性，如理解系统做什么、跟踪流经系统的数据流、定位系统的复杂部分。尽管不像人们通常所想像的那样，对问题进行划分会奇迹般地把一个复杂的问题转换成一组简单问题的集合，然而，模块化允许我们孤立问题中最难以把握的部分，结果是防止我们被不相关的功能和数据所困扰或导入歧途。

2. 抽象(Abstraction)

抽象是人类在认识世界和改造世界的实践中所普遍采用的原则和方法。现实世界中的事物是复杂多样的，在解决问题的过程中，集中考虑和当前问题有关的方面，而忽略和当前问题无关的方面，这就是抽象。或者说抽象就是抽出事物的本质特性而暂时不考虑它们的细节。

在软件模块结构图中，下层模块是对上层模块的细化，我们认为顶层模块的抽象程度最高，当我们的注意力转到较低的层次时，可以发现关于每个模块更多的细节，即模块是按照不同的抽象级别安排的。抽象级别有助于我们理解系统所要解决的问题，通过自顶向下地检查各层模块，抽象的问题逐渐细化为相应解决方案的细节描述。从一定意义上说，高层抽象模块向读者隐藏了功能实现的细节，这就是信息隐蔽。再进一步，模块之间相互隐藏自身的实现细节对一个好的设计来说是至关重要的。

抽象和信息隐蔽允许我们在总体设计中检查模块间的关联方式。一个模块同其它模块的独立程度是评价一个设计好坏的重要度量尺度。独立性要求有两方面的原因，首先，当一个模块的功能不是同其它模块紧密地联系在一起时，比较容易理解；其次，一个模块独立于其它模块时比较容易修改。需求和设计的改变经常意味着一些模块必须被修改，修改影响到功能或数据或两者兼有之。如果模块之间相互紧密依赖，那么对一个模块的修改影响到其它模块的几率也就越大。模块越独立，它就越容易同受改变影响的模块隔离开来。

下面关于耦合和内聚的两条准则可以帮助我们评价模块的独立程度及设计质量。

3. 耦合(Coupling)

耦合是对不同模块之间相互依赖程度的度量。紧密耦合是指两个模块之间存在着很强的依赖关系；松散耦合是指两个模块之间存在一些依赖关系，但它们之间的连接比较弱；无耦合是指模块之间根本没有任何连接(见图 4.20)。耦合的强度依赖于以下几个因素：

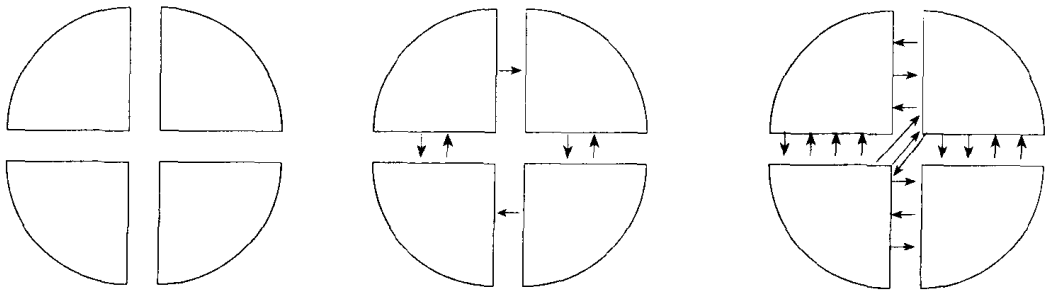


图4.20　模块间的耦合程度

· 一个模块对另一个模块的引用，例如，模块 A 调用模块 B，那么模块 A 的功能依赖于模块 B 的功能；

· 一个模块向另一个模块传递的数据量，例如，模块 A 为了完成其功能需要模块 B 向其传递一组数据，那么模块 A 依赖于模块 B；

· 一个模块施加到另一个模块的控制的数量，例如，模块 A 传递给模块 B 一个控制信号，模块 B 执行的操作依赖于控制信号的值；

· 模块之间接口的复杂程度，例如，如果模块 A 给模块 B 传递一个简单的数值，但模块 C 和模块 D 之间传递的是数组，甚至是控制信号，则模块 A 和 B 之间接口的复杂度小于模块 C 和 D 之间的接口复杂度。

下面我们按从强到弱的顺序给出几种常见的模块间耦合的类型：

(1) 内容耦合

当一个模块直接修改或操作另一个模块的数据时，就发生了内容耦合。被修改的模块完全依赖于修改它的模块，主要表现为以下两种情形：

① 一个模块访问或修改另一个模块的内部数据；

② 一个模块不通过正常入口而跳转到另一个模块的内部。

内容耦合是最高程度的耦合，应该尽量避免使用。

(2) 公共耦合

我们可以通过使用全局或公共数据来多少降低一些耦合强度(如 FORTRAN 语言的 COMMAN 块或其它高级程序设计语言中声明的全局变量)，这种两个以上的模块共同引用一个全局数据项就称为公共耦合。模块间的依赖关系依旧存在，因为对全局数据项的修改作用于所有访问该数据项的模块。在具有大量公共耦合的结构中，确定究竟是哪个模块给全局变量赋了一个特定的值是十分困难的。公共耦合的情形见图 4.21。

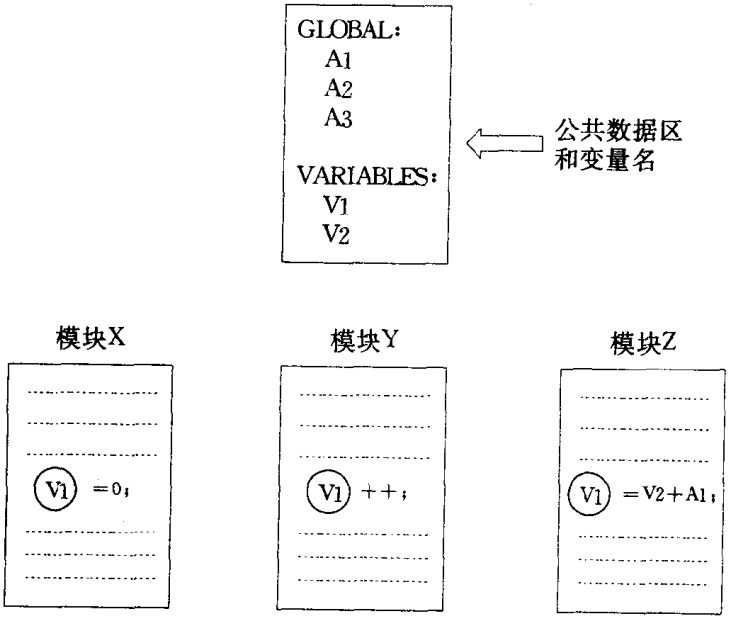


图4.21 公共耦合示意图

(3) 控制耦合

一个模块在界面上传递一个信号控制另一个模块，接收信号的模块的动作根据信号值进行调整，称为控制耦合。通过保证每个模块只完成一个特定的功能，可以大大地减少模块间传递的控制信息。

(4) 标记耦合

若两个模块至少有一个通过界面传递的公共参数包含内部结构，如字符串或记录，则称这两个模块之间存在标记耦合。标记耦合的例子见图 4.22。

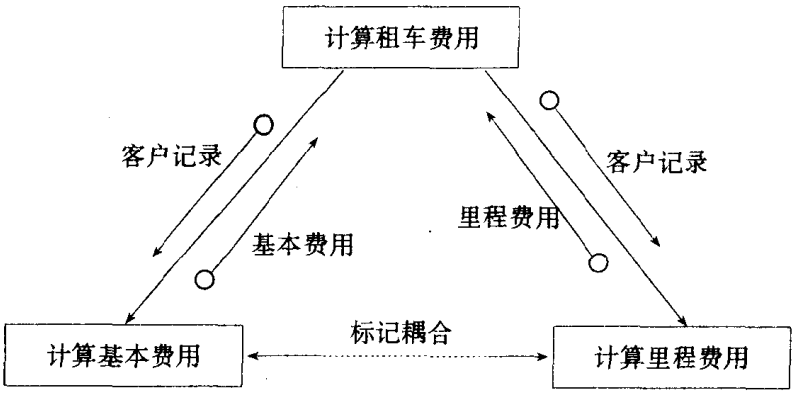


图4.22 标记耦合的例子

(5) 数据耦合

模块间通过参数传递基本类型的数据，称为数据耦合。数据耦合是最简单的耦合形式，系统中至少必须存在这种类型的耦合，因为只有当某些模块的输出数据作为另一些模块的输入数据时，模块之间才能联结成为一个整体，从而完成有意义的功能。耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上我们应采取以下原则：如果模块间必须存在耦合，就尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，坚决避免使用内容耦合。

4. 内聚(Cohension)

不同于度量模块之间的相互依赖程度，内聚度量的是一个模块内部各成分之间相互关联的强度。一个模块内聚程度越高，该模块内部各成分之间以及同模块所完成的功能之间的关联也就越强。换句话说，如果一个模块的所有成分都直接参与并且对于完成同一功能来说都是最基本的，则该模块是高内聚的。正如耦合分成不同的级别，内聚也是如此。设计时追求的目标应尽量使每个模块做到高内聚，这样模块的各个成分都和模块的单一功能直接相关。以下从低到高给出一些常见的内聚类型，如图 4.23 所示。

(1) 偶然内聚

如果一个模块的各成分之间毫无关系，则称为偶然内聚。例如，有时在编写一段程序时，发现有一组语句在两处或多处出现，于是把这组语句作为一个模块以减少书写工作量，如果这组语句彼此间没有任何关系，这时就出现了偶然内聚。

在偶然内聚的模块中，各个成分之间没有实质性联系，很可能在一个应用场合需要修改这个模块，在另一个应用场合又不允许这种修改，从而陷入困境。事实上，偶然内聚的模块出现修改错误的概率比其它类型的模块高得多。

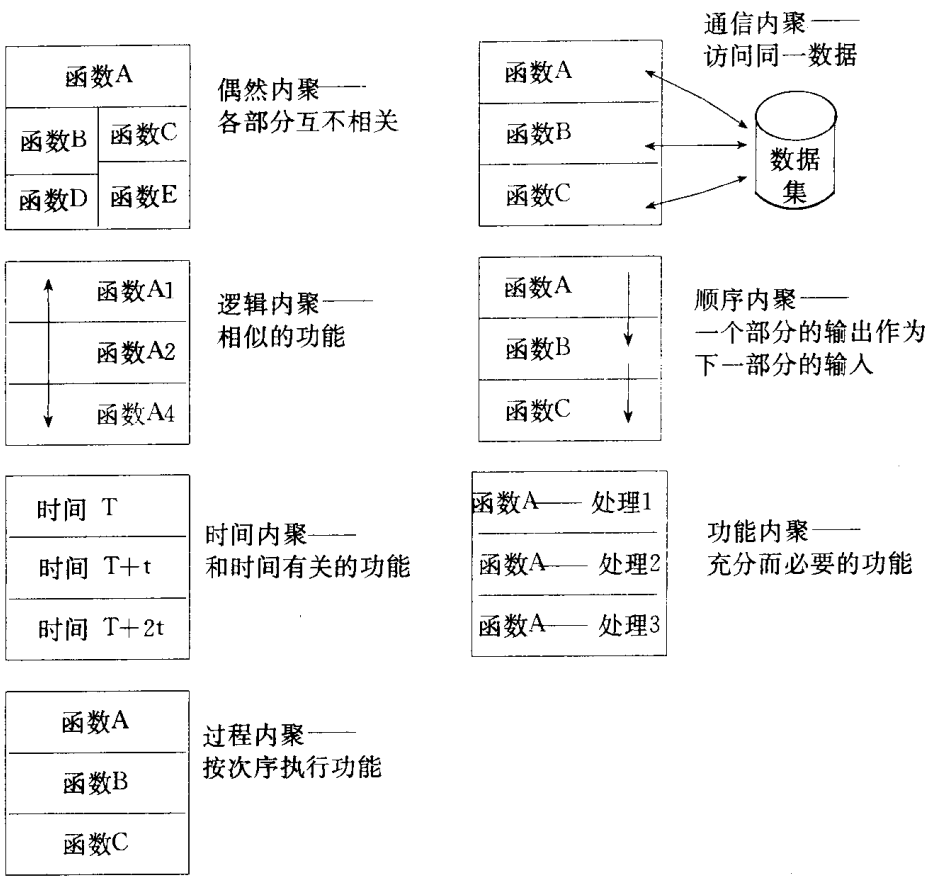


图4.23　一些常见的内聚类型

(2) 逻辑内聚

几个逻辑上相关的功能被放在同一模块中，则称为逻辑内聚。例如，一个模块读取各种不同类型外设的输入(包括卡片、磁带、磁盘、键盘等)，而不管这些输入从哪儿来、做什么用，因为这个模块的各成分都执行输入，所以该模块是逻辑内聚的。

尽管逻辑内聚比偶然内聚合理一些，但逻辑内聚的模块各成分在功能上并无关系，即使局部功能的修改有时也会影响全局，因此这类模块的修改也比较困难。例如在上面的例子中，因为输入可以用于不同模块的不同目的，所以实质上是把不同的功能放在一个地方。

(3) 时间内聚

如果一个模块完成的功能必须在同一时间内执行(例如，初始化系统或一组变量)，但这些功能只是因为时间因素关联在一起，则称为时间内聚。

时间内聚在一定程度上反映了系统的某些实质，因此比逻辑内聚好一些。

(4) 过程内聚

如果一个模块内部的处理成分是相关的，而且这些处理必须以特定的次序执行，则称为过程内聚。使用程序流程图作为工具设计软件时，常常通过研究流程图确定模块的划分，这样得到的往往是过程内聚的模块。

(5) 通信内聚

如果一个模块的所有成分都操作同一数据集或生成同一数据集，则称为通信内聚。例如，所有的处理都在一个磁盘或磁带上实施，有时这样的安排显得很方便。然而，通信内聚经常破

坏设计的模块化和功能独立性。

(6) 顺序内聚

如果一个模块的各个成分和同一个功能密切相关，而且一个成分的输出作为另一个成分的输入，则称为顺序内聚。因为模块不是基于功能关系组织在一起的，很可能一个模块没有包含对于完成一个功能所需的全部处理。

(7) 功能内聚

最理想的内聚是功能内聚，模块的所有成分对于完成单一的功能都是基本的。功能内聚的模块对完成其功能而言是充分必要的。

内聚和耦合是密切相关的，同其它模块存在强耦合的模块常意味着弱内聚，而强内聚的模块常意味着该模块同其它模块之间松散的耦合。在进行软件设计时，应力争做到强内聚、弱耦合。

4.5 启发式规则

除了上节介绍的在设计中必须遵守的准则之外，人们在开发软件的长期实践中还积累了丰富的经验，总结这些经验得出了一些启发式规则，这些规则在许多场合能给设计人员以有益的启示，遵循这些规则有助于改善软件结构，得到高质量的软件。下面我们就介绍几条启发式规则。

1. 改进软件结构提高模块独立性

设计出软件的初步结构以后，应该审查分析这个结构，通过模块分解或合并，力求降低耦合提高内聚。例如，多个模块公有的一个子功能可以独立成一个模块，供这些模块调用；有时可以通过分解或合并模块以减少控制信息的传递及对全局数据的引用，并且降低接口的复杂程度。

2. 模块规模应该适中

经验表明，一个模块的规模不应过大，最好能写在一页纸内(通常不超过 60 行语句)。有人从心理学角度研究得知，当一个模块包含的语句数超过 30 以后，模块的可理解程度迅速下降。

过大的模块往往是由于分解不充分，但是进一步分解必须符合问题结构，一般说来，分解不应该以降低模块独立性为代价。

过小的模块开销大于有效操作，而且模块数目过多将使系统接口复杂，因此过小的模块有时不值得单独存在，特别是当只有一个模块调用它时，通常可以把它合并到上级模块中去而不必单独存在。

3. 深度、宽度、扇出和扇入应适中

深度表示软件结构中控制的层数，它往往能粗略地标志一个系统的大小和复杂程度。深度和程序长度之间应该有粗略的对应关系，当然这个对应关系是在一定范围内变化的。如果层数过多则应该考虑是否许多管理模块过分简单了，能否适当合并。

宽度是软件结构中同一个层次上的模块总数的最大值。一般说来，宽度越大系统越复杂。对宽度影响最大的因素是模块的扇出。

扇出是一个模块直接控制(调用)的下级模块数目，扇出过大往往意味着模块过分复杂，需要控制和协调过多的下级模块；扇出过小(例如总是 1)意味着功能过分集中，也会导致复杂的

模块。经验表明，一个设计得好的典型系统的平均扇出通常是 3 或 4（扇出的上限通常是 5—9）。

扇出太大一般是因为缺乏中间层次，应该适当增加中间层次的控制模块。扇出太小时可以把下级模块进一步分解成若干个子功能模块，或者合并到它的上级模块中去。当然，分解模块或合并模块必须符合问题结构，不能违背模块独立性原则。

一个模块的扇入表明有多少个上级模块直接调用它，扇入越大则共享该模块的上级模块数目越多，这是有好处的，但是，不能违背模块独立性原则单纯追求高扇入。

观察大量软件系统后发现，设计得很好的软件结构通常顶层扇出比较高，中层扇出较少，底层扇入到公共的实用模块中去（底层模块有高扇入）。即系统的模块结构呈现为“葫芦”形状。

4. 模块的作用域应该在控制域之内

模块的作用域定义为受该模块内一个判定影响的所有模块的集合。模块的控制域是这个模块本身以及所有直接或间接从属于它的模块的集合。例如，在图 4.24 中模块 A 的控制域是 A，B，C，D，F，E 等模块的集合。

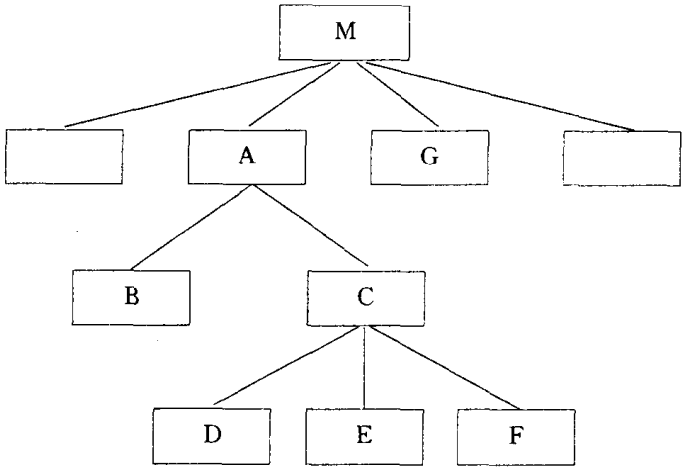


图 4.24　模块的作用域和控制域

在一个设计得很好的系统中，所有受判定影响的模块应该都从属于作出判定的那个模块，最好局限于作出判定的那个模块本身及它的直属下级模块。例如，如果图 4.24 中模块 A 做出的判定只影响模块 B，那么是符合这条规则的。但是，如果模块 A 做出的判定同时还影响模块 G 中的处理过程，又会有什么坏处呢？首先，这样的结构使得软件难于理解；其次，为了使得 A 中的判定能影响 G 中的处理过程，通常需要在 A 中给一个标记设置状态以指示判定的结果，并且应该把这个标记传递给 A 和 G 的公共上级模块 M，再由 M 把它传给 G。这个标记是控制信息而不是数据，因此将使模块间出现控制耦合。

怎样修改软件结构才能使作用域是控制域的子集呢？一个方法是把做判定的点往上移，例如，把判定从模块 A 中移到模块 M 中。另一个方法是把那些在作用域内但不在控制域内的模块移到控制域内，例如，把模块 G 移到模块 A 的下面，成为它的直属下级模块。到底采用哪种方法改进软件结构，需要根据具体问题统筹考虑。一方面应该考虑哪种方法更现实，另一方面应该使软件结构能更好地体现问题本来的结构。

5. 力争降低模块接口的复杂性

模块接口复杂是软件发生错误的一个主要原因。应该仔细设计模块接口，使得信息传递简单并且和模块的功能一致。

例如，求一元二次方程的根的模块 QUAD-ROOT(TBL,X)，其中用数组 TBL 传送方程的系数，用数组 X 回送求得的根。这种传递信息的方法不利于对这个模块的理解，不仅在维护期间容易引起混淆，在开发期间也可能发生错误。下面这种接口可能是比较简单的：

QUAD-ROOT(A,B,C,ROOT1,ROOT2)

其中 A,B,C 是方程的系数，ROOT1 和 ROOT2 是算出的两个根。

接口复杂或不一致(即看起来传递的数据之间没有联系)，是紧耦合或低内聚的征兆，应该重新分析这个模块的独立性。

6. 模块功能应该可以预测

模块的功能应该能够预测，但也要防止模块功能过分局限。

如果一个模块可以当做一个黑盒子，也就是说，只要输入的数据相同就产生同样的输出，这个模块的功能就是可以预测的。带有内部状态的模块的功能可能是不可预测的，因为它的输出可能取决于所处的状态，由于内部状态对于上级模块而言是不可见的，所以这样的模块既不易理解又难于测试和维护。

如果一个模块只完成一个单独的子功能，则呈现高内聚；但是，如果一个模块任意限制局部数据结构的大小，过分限制在控制流中可以做出的选择或者外部接口的模式，那么这种模块的功能就过分局限，使用范围也就过分狭窄了。在使用过程中将不可避免地需要修改功能过分局限的模块，以提高模块的灵活性，扩大它的使用范围；但是，在使用现场修改软件的代价是很高的。

以上列出的启发式规则多数是经验规律，对改进软件设计，提高软件质量，往往有重要的参考价值；但是，它们既不是设计的目标也不是设计时应该普遍遵循的原理。

4.6 设 计 优 化

在上面第 4.3 节“总体设计的方法”中，我们把信息流分为变换流和事物流两种，针对不同的信息流类型有不同的设计方法，而每种方法又分成若干设计步骤，在每一个设计步骤中都应该考虑好的设计的准则和启发式规则，尽管这样，仍需要根据模块独立性原则进行最终的优化，也就是上面第 4.3 节“设计步骤”中的第 7 步的工作。

对初步分解得到的软件结果，总可以根据模块独立性原则进行精化。为了产生合理的分解，得到尽可能高的内聚、尽可能松散的耦合，最重要的是，为了得到一个易于实现、易于测试和易于维护的软件结构，应该对初步分解得到的模块进行再分解或合并。

还是以上面的数字仪表板为例，对于从前面的设计步骤得到的软件结构，还可以做许多修改。下面是某些可能的修改：

- 输入结构中的模块“转换成 rpm”和“收集 sps”可以合并；
- 模块“确定加速/减速”可以放在模块“计算 mph”下面，以减少耦合；
- 模块“加速/减速显示”可以放在模块“显示 mph”的下面。

经过上述修改后的软件结构画在图 4.25 中。

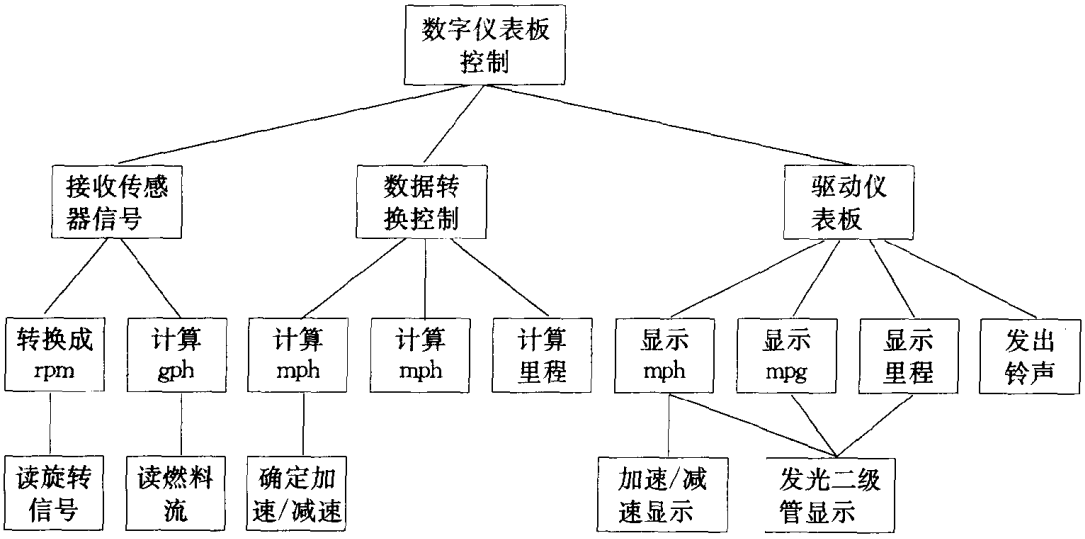


图4.25　精化后的数字仪表板系统的软件结构

考虑设计优化问题时应该记住，“一个不能工作的‘最佳设计’的价值是值得怀疑的”。软件设计人员应该致力于开发能够满足所有功能和性能要求，而且按照设计原理和启发式设计规则衡量是值得接受的软件。

应该在设计的早期阶段尽量对软件结构进行精化，可以导出不同的软件结构，然后对它们进行评价和比较，力求得到“最好”的结果。这种优化的可能，是把软件结构设计和过程设计分开的真正优点之一。

注意，结构简单通常既表示设计风格优雅，又表明效率高。设计优化应该力求做到在有效的模块化的前提下使用最少量的模块，以及在满足信息要求的前提下使用最简单的数据结构。

对于时间是决定性因素的应用场合，可能有必要在详细设计阶段，也可能在编写程序的过程中进行优化。软件开发人员应该认识到，程序中相对说比较小的部分（典型的10%—20%），通常占用全部处理时间的大部分（50%—80%）。用下述方法对时间起决定性作用的软件进行优化是合理的：

① 在不考虑时间因素的前提下开发并精化软件结构；

② 在详细设计阶段选出最耗费时间的那些模块，仔细地设计它们的处理过程（算法），以求提高效率；

③ 使用高级程序设计语言编写程序；

④ 在软件中孤立出那些大量占用处理机资源的模块；

⑤ 必要时重新设计或用依赖于机器的语言重写上述大量占用资源的模块的代码，以求提高效率。

上述优化方法遵守了一句格言：“先使它能工作，然后再使它快起来。”

4.7 ××××××系统软件设计说明书

1. 引言

1.1 编写目的

说明编写本软件设计说明书的目的。

1.2 背景说明

(1) 给出待开发的软件产品的名称；

(2) 说明本项目的提出者，开发者及用户；

(3) 说明该软件产品将做什么，如有必要，说明不做什么。

1.3 术语定义

列出本文档中所用的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出本文档中所引用的全部资料，包括标题、文档编号、版本号、出版日期及出版单位等，必要时注明资料来源。

2. 总体设计

2.1 需求规定

说明对本软件的主要输入、输出、处理的功能及性能要求。

2.2 运行环境

简要说明对本软件运行的软件、硬件环境和支持环境的要求。

2.3 处理流程

说明本软件的处理流程，尽量使用图、文、表的形式。

2.4 软件结构

在DFD图的基础上，用模块结构图来说明各层模块的划分及其相互关系，划分原则上应细到程序级(即程序单元)，每个单元必须执行单独一个功能(即单元不能再分了)。

3. 运行设计

3.1 运行模块的组合

说明对系统施加不同的外界运行控制时所引起的各种不同的运行模块的组合，说明每种运行所经历的内部模块和支持软件。

3.2 运行控制

说明各运行控制方式、方法和具体的操作步骤。

4. 系统出错处理

4.1 出错信息

简要说明每种可能的出错或故障情况出现时，系统输出信息的格式和含义。

4.2 出错处理方法及补救措施

说明故障出现后可采取的措施，包括：

(1) 后备技术。当原始系统数据万一丢失时启用的副本的建立和启动的技术，如周期性的信息转储；

(2) 性能降级。使用另一个效率稍低的系统或方法(如手工操作、数据的人工记录等)，以求得到所需结果的某些部分；

(3) 恢复和再启动。用建立恢复点等技术，使软件再开始运行。

5. 模块设计说明

以填写模块说明表的形式，对每个模块给出下述内容：

(1) 模块的一般说明，包括名称、编号、设计者、所在文件、所在库、调用本模块的模块名和本模块调用的其它模块名；

(2) 功能概述；

(3) 处理描述，使用伪码描述本模块的算法、计算公式及步骤；

(4) 引用格式；

(5) 返回值；

(6) 内部接口，说明本软件内部各模块间的接口关系，包括：

(a) 名称，

(b) 意义，

(c) 数据类型，

(d) 有效范围，

(e) I/O 标志；

(7) 外部接口，说明本软件同其它软件及硬件间的接口关系，包括：

(a) 名称，

(b) 意义，

(c) 数据类型，

(d) 有效范围，

(e) I/O 标志，

(f) 格式，指输入或输出数据的语法规则和有关约定，

(g) 媒体；

(8) 用户接口，说明将向用户提供的命令和命令的语法结构，以及软件的回答信息，包括：

(a) 名称，

(b) 意义，

(c) 数据类型，

(d) 有效范围，

(e) I/O 标志，

(f) 格式，指输入或输出数据的语法规则和有关约定，

(g) 媒体。

附:模块说明表

模块说明表

制表日期:　　年　月　日

模块名:	模块编号:	设计者:
模块所在文件:	模块所在库:	
调用本块的模块名:		
本模块调用的其它模块名:		
功能概述:		
处理描述:		
引用格式:		
返回值:		

	名　称	意　义	数据类型	数值范围	I/O 标志
内部接口					

	名　称	意　义	数据类型	I/O 标志	格　式	媒　体	
外部接口							
用户接口							

第五章 详细设计

经过总体设计阶段的工作，已经确定了软件的模块结构和接口描述，但这时每个模块仍处于黑盒子级。详细设计阶段的根本目标是确定怎样具体地实现所要求的系统，也就是说，经过这个阶段的设计工作，应该得出对目标系统的精确描述，从而在编码阶段可以将这个描述直接翻译成用某种程序设计语言书写的程序。因此，详细设计的结果基本上决定了最终的程序代码的质量。

详细设计以总体设计阶段的工作为基础，但又不同于总体设计阶段，主要表现为以下两个方面：

① 在总体设计阶段，数据项和数据结构以比较抽象的方式描述，例如，总体设计可以声明一组值从概念上表示一个矩阵，详细设计就要确定用什么数据结构来实现这样的矩阵，比如特殊的稀疏矩阵技术也许是最合适的。

② 详细设计要提供关于算法的更多的细节，例如，总体设计可以声明一个模块的作用是对一个表进行排序，详细设计则要确定使用哪种排序算法。在详细设计阶段为每个模块增加了足够的细节，使得程序员能够以相当直接的方式编码每个模块。

因此，详细设计的模块包含实现对应的总体设计的模块所需要的处理逻辑，主要有：

① 详细的算法

② 数据表示和数据结构

③ 实施的功能和使用的数据之间的关系

每个模块被编码成过程、子程序、函数或其它类型的命名实体。

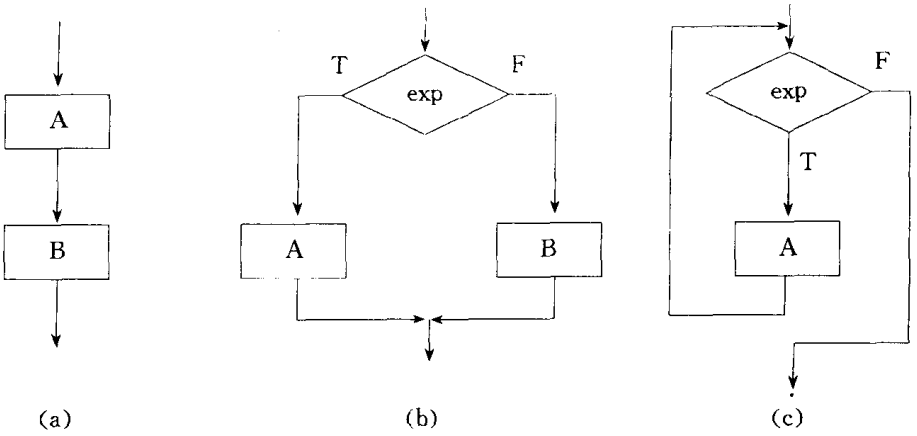
5.1 结构化程序设计

结构化程序设计的概念最早由 E. W. Dijkstra 在 60 年代中期提出，并在 1968 年著名的 NATO 软件工程会议上首次引起人们的广泛关注。1966 年，C. Bohm 和 G. Jacopini 在数学上证明了，只用三种基本的控制结构就能实现任何单入口单出口的程序，这三种基本的控制结构就是“顺序”、“选择”和“循环”，它们的流程图表示见图 5.1。实际上，用顺序结构和循环结构（又称 DO-WHILE 结构）完全可以实现选择结构（又称 IF-THEN-ELSE 结构），因此，理论上最基本的控制结构只有两种。

Bohm 和 Jacopini 的证明给结构化程序设计技术奠定了理论基础。

与此同时，结构化程序设计技术作为一种新的程序设计思想、方法和风格，也开始引起工业界的重视。1971 年，IBM 公司在纽约时报信息库管理系统的设计中使用了结构化程序设计技术，获得了巨大的成功，于是开始在整个公司内部全面采用结构化程序设计技术，并介绍给了它的许多用户。IBM 在计算机界的影响为结构化程序设计技术的推广起到了推波助澜的作用。

那么，什么是结构化程序设计技术呢？一种比较流行的定义是：



(a) 顺序结构，先执行A再执行B；(b) IF - THEN - ELSE型选择(分支)结构；(c) DO - WHILE型循环结构。

图5.1 三种基本的控制结构

结构化程序设计技术是一种程序设计技术，它采用自顶向下逐步求精的设计方法和单入口单出口的控制结构，并且只包含顺序、选择和循环三种结构。

结构化程序设计的目标之一是使程序的控制流程线性化，即程序的动态执行顺序符合静态书写结构，这就增强了程序的可读性，不仅容易理解、调试、测试和排错，而且给程序的形式化证明带来了方便。正如 Bohm 和 Jacopini 所证明的，顺序结构、选择结构和循环结构构成了结构化程序设计的核心，它们组合使用可以实现任意复杂的处理逻辑，除此之外无需其它控制结构，这样一来，无条件转移指令 GOTO 语句变得多余了。

1968 年，ACM 通信发表了 Dijkstra 的短文“GOTO Statement considered harmful”，引起了人们的极大关注。Dijkstra 认为：GOTO 语句太原始，是构成程序混乱不堪的罪魁祸首，GOTO 语句应该从一切高级程序设计语言中消失掉。自此开始了一场是否消灭 GOTO 语句的旷日持久的争论。

关于 GOTO 语句的争论直到 1974 年以后才逐渐平息下来。经过讨论人们达成了共识，GOTO 语句并没有完全从高级语言中消失，现代的程序设计语言，如 PASCAL，C，Ada 中仍保留了 GOTO 语句就是一个证明，但要限制 GOTO 语句的使用范围。我们还是引用 1974 年 N. Wirth 在《PASCAL User Manual and Report》一书中关于 GOTO 语句的建议：

当出现算法的自然结构被破坏的异常情况时，应保留 GOTO 语句。一个好的原则是避免使用跳转表达正常的循环或条件语句，因为这样的跳转破坏了程序的静态文本结构在动态计算结构中的反映。换句话说，如果程序的静态结构和动态结构没有较好对应起来，就会影响程序的清晰度，并使验证工作变得更加困难。

这场关于 GOTO 语句争论的实质在于：程序设计首先是讲究结构，还是讲究效率。好结构的程序不一定是效率最高的程序。结构化程序设计的观点是要求设计好结构的程序。在计算机硬件技术迅速发展的今天，人们已普遍认为，除了系统的核心程序部分以及其它一些有特殊要求的程序以外，在一般情况下，宁可牺牲一些效率，也要保证程序有一个好的结构。

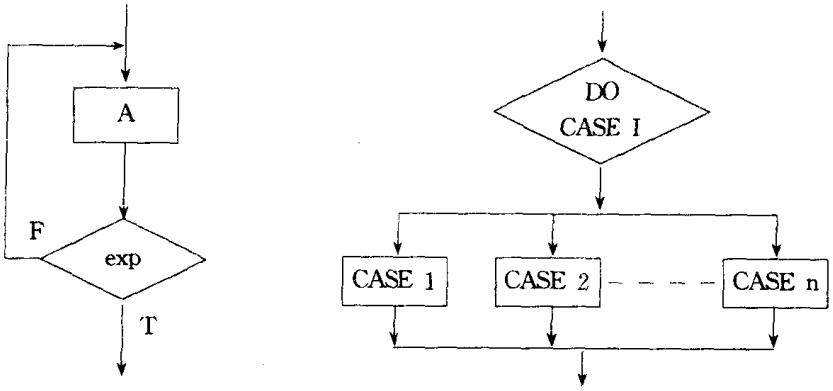
5.2 详细设计的工具

详细设计的任务是给出软件模块结构中各个模块的内部过程描述，也就是模块内部的算法设计。我们这里并不打算讨论具体模块的算法设计（感兴趣的读者可以参考N. Wirth 所著的《Algorithms + Data Structures = Programs》一书），而是讨论这些算法的表示形式。详细设计的工具可以分为图形、表格和语言三种，无论是哪类工具，对它们的基本要求都是能提供对设计的无歧义的描述，即能指明控制流程、处理功能、数据组织以及其它方面的实现细节，从而在编码阶段能把设计描述直接翻译成程序代码。下面我们介绍一些典型的详细设计工具。

5.2.1 程序流程图

程序流程图又称为程序框图，它是历史最悠久、使用最广泛的描述软件设计的方法，然而它也是用得最混乱的一种方法。从40年代末到70年代中期，程序流程图一直是软件设计的主要工具。它的主要优点是对控制流程的描绘很直观，便于初学者掌握。由于流程图历史悠久，广泛为人所熟悉，所以尽管它有种种缺点，甚至许多人建议停止使用它，但至今仍在广泛使用着。不过总的趋势是越来越多的人不再使用程序流程图了。

程序流程图中使用的主要符号包括顺序结构、选择结构和循环结构，如上节图5.1所示。值得注意的是，程序流程图中的箭头代表的是控制流而不是数据流，这一点是同数据流图中的箭头不同的。除了以上的三种基本控制结构外，为了方便起见，程序流程图中还经常使用其它一些等价的符号，如图5.2所示。



(a) REPEAT - UNTIL型循环结构；　　(b) 多分支结构。

图5.2 其它常用的控制结构

程序流程图的主要缺点如下：

① 程序流程图本质上不是逐步求精的好工具，它诱使程序员过早地考虑程序的控制流程，而不去考虑程序的全局结构；

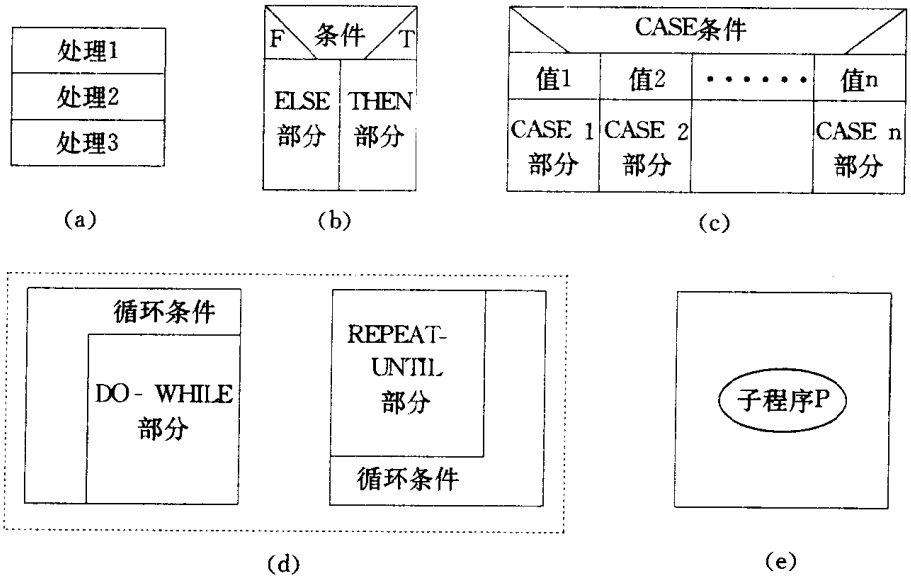
② 程序流程图中用箭头代表控制流，因此程序员不受任何约束，可以完全不顾结构程序设计的精神，随意转移控制；

③ 程序流程图不易表示数据结构。

应该指出，详细的微观程序流程图——每个符号对应于源程序的一行代码，对于提高大型系统的可理解性作用甚微。

5.2.2 盒图(N-S 图)

出于要有一种不允许违背结构程序设计精神的考虑，在 70 年代早期，Nassi 和 Shneiderman 提出了盒图，又称为 N-S 图。同程序流程图相比，它以一种结构化的方式严格地限制从一个处理到另一个处理的控制转移。



(a) (b) (c)

(d) (e)

(a) 顺序；(b) IF - THEN - ELSE型分型；(c) CASE型多分支；
(d) 循环；(e) 调用子程序P。

图5.3 盒图的基本符号

每一个 N-S 图开始于一个大的矩形，表示它所描述的模块。该矩形的内部被分成不同的部分，分别表示不同的子处理过程，这些子处理过程又可以进一步分解成更小的部分。由于每次分解都只能使用图 5.3 给出的基本符号，因此最终得到的详细设计必然是结构化的。

5.2.3 PAD 图

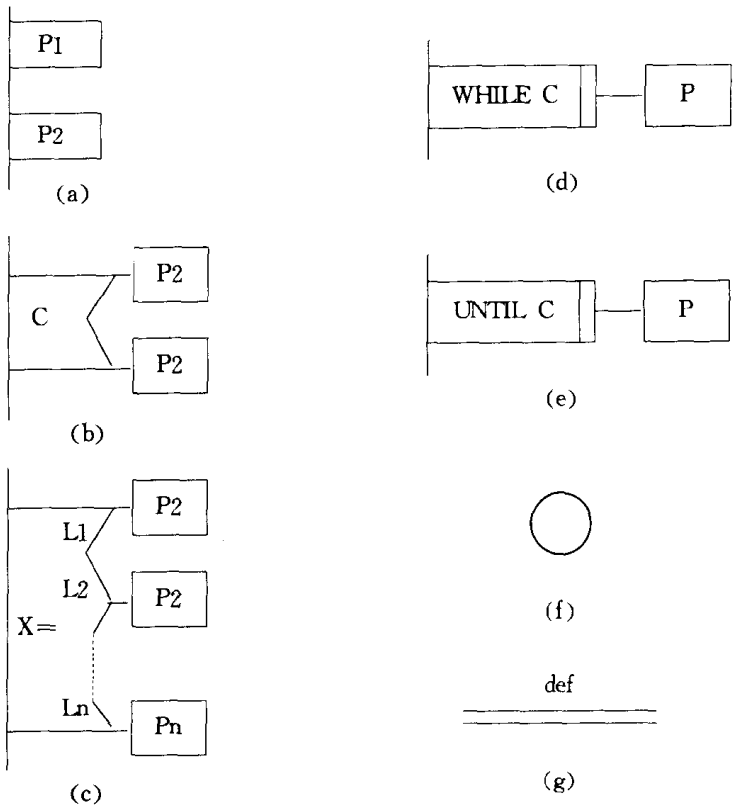
PAD 是问题分析图(Problem Analysis Diagram)的英文缩写，自 1973 年由日本日立公司发明以来，已经得到一定程度的推广。它用二维树形结构的图表示程序的控制流，将这种图转换为程序代码比较容易。图 5.4 给出 PAD 图的基本符号。

PAD 图的主要优点如下：

① 使用表示结构化控制结构的 PAD 符号所设计出来的程序必然是结构化程序；

② PAD 图所描述的程序结构十分清晰。图中最左边的竖线是程序的主线，即第一层控制结构。随着程序层次的增加，PAD 图逐渐向右延伸，每增加一个层次，图形向右扩展一条竖线。PAD 图中竖线的总条数就是程序的层次数；

③ 用 PAD 图表现程序逻辑，易读、易懂、易记。PAD 图是二维树型结构的图形，程序从图中最左边上端的结点开始执行，自上而下，从左向右顺序执行；



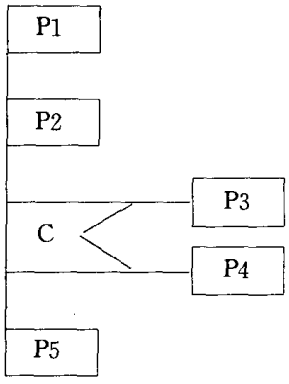
(a) 顺序(先执行P1后执行P2)；(b) 选择(IF C THEN P1 ELSE P2)；
(c) CASE型多分支；(d) WHILE型循环(WHILE C DO P)；
(e) UNTIL型循环(REPEAT P UNTIL C)；(f) 语句标号；(g) 定义。

图5.4 PAD图的基本符号

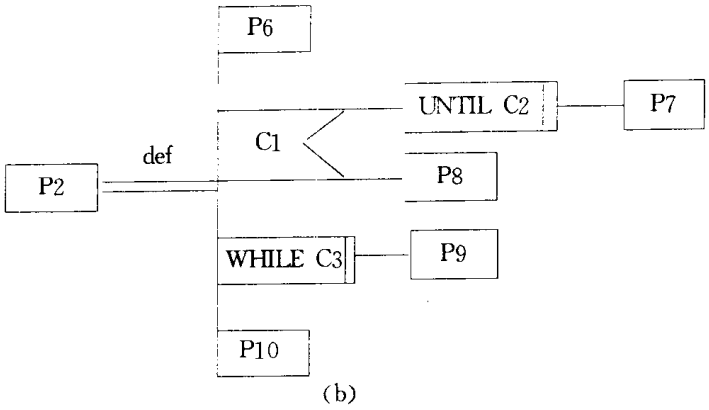
④ 很容易将PAD图转换成高级语言源程序，这种转换可用软件工具自动完成，从而可省去人工编码的工作，有利于提高软件可靠性和软件生产率；

⑤ 既可用于表示程序逻辑，也可用于描述数据结构；

⑥ PAD图的符号支持自顶向下、逐步求精方法的使用。开始时设计者可以定义一个抽象程序，随着设计工作的深入而使用"def"符号逐步增加细节，直至完成详细设计，如图5.5所示。



(a)



(a) 初始的PAD图；(b) 使用def符号细化处理P2

图5.5 使用PAD图提供的定义功能来逐步求精的例子

PAD 图是面向高级程序设计语言的，为FORTRAN，COBOL，PASCAL 和 C 等常用的高级程序设计语言都提供了一整套相应的图形符号。由于每种控制语句都有一个图形符号与之对应，显然将 PAD 图转换成与之对应的高级语言程序比较容易。

5.2.4 类程序设计语言(PDL)

类程序设计语言(Program Design Language，简称 PDL)也称为伪码，这是一个笼统的名称，现在有许多种不同的 PDL 在使用。它是用正文形式表示数据结构和处理过程的设计工具。

一般说来，PDL 是一种"混合"语言，一方面，PDL 具有严格的关键字外部语法，通常是借用某种结构化的程序设计语言(如 PASCAL 或 C)的语法控制框架，用于定义控制结构和数据结构；另一方面，PDL 使用一种语言(通常是某种自然语言，如汉语或英语)的词汇，灵活自由地表示实际操作和判定条件，以便可以适应各种工程项目的需要。

PDL 具有下述特点：

① 关键字的固定语法，提供了结构化控制结构、数据说明和模块化的手段。为了使结构清晰和可读性好，通常在所有可能嵌套使用的控制结构的头和尾都有关键字，例如，if…fi(或 endif)等等；

② 自然语言的自由语法，用于描述处理过程和判定条件；

③ 数据说明的手段，既包括简单的数据结构(例如纯量和数组)，又包括复杂的数据结构(例如，链表或层次的数据结构)；

④ 模块定义和调用的技术，提供各种接口描述模式。

PDL 作为一种设计工具有如下一些优点：

① 可以作为注释直接插在源程序中间。这样做能促使维护人员在修改程序代码的同时也相应地修改 PDL 注释，因此有助于保持文档和程序的一致性，提高了文档的质量；

② 可以使用普通的正文编辑程序或文字处理系统，很方便地完成 PDL 的书写和编辑工作；

③ 已经有自动处理程序存在，而且可以自动由 PDL 生成程序代码。

PDL 的缺点是不如图形工具形象直观，描述复杂的条件组合与动作间的对应关系时，不如判定表或判定树清晰简单。

除以上介绍的外，详细设计工具还包括我们前面介绍过的 IPO 图、判定树和判定表等。

第六章　面向对象分析

软件工程的概念是在60年代末期提出的，随后提出的结构化方法作为软件开发的主流方法，曾被广泛地应用于各种软件项目的开发。它的研究和发展帮助人们更加清楚地认识了软件开发的实质，并成功地支持了一些大型项目的开发，对于解决软件危机起到了一定的缓解作用，但远未充分解决软件危机。究其原因在于：用Von Neumann机所求解问题的问题空间结构同Von Neumann机所用的求解问题方法的方法空间结构的不一致性。而结构化方法从本质上看仍具有Von Neumann机体系结构的特点，是软件开发人员从开发软件的立场出发而确定的，并不是从人们认识客观世界的过程和方法出发的。

同人们认识世界的一般规律一样，面向对象方法学认为：客观世界是由许多各种各样的对象所组成的，每种对象都有各自的内部状态和运动规律，不同对象间的相互作用和联系就构成了各种不同的系统，构成了我们所面对的五彩缤纷的世界。面向对象方法学追求的目标是使解决问题的方法空间同客观世界的问题空间结构达到一致，对于软件工程濒临的困境和人工智能所遇到的障碍都是很有希望的突破口之一，必将成为90年代软件开发的主流方法。

6.1　面向对象技术概述

6.1.1　面向对象技术的历史、现状和发展

一般认为，面向对象方法起源于60年代末出现的Simula语言。在这个语言中引入了数据抽象和类的概念，但真正为面向对象程序设计奠定基础的是Smalltalk语言，“面向对象”这个词也是Smalltalk首先采用的。在Smalltalk中一切都是对象，Smalltalk的目标是使得软件尽可能以“自治”的单元来设计。Smalltalk不仅仅是一门语言，更是一个完整的程序设计环境，其中包括四个部分：

① 程序设计语言的核心，包括语言的语法和语义；

② 程序设计风格，使用核心生成软件系统的方法，这是和面向对象程序设计结合最密切的部分；

③ 程序设计系统，是对象和类的集合，为编程提供方便；

④ 用户界面模型，提供对象和类结合的样式。

Smalltalk被认为是第一个真正的面向对象程序设计语言，直到今天仍被认为是最纯的面向对象语言之一。Smalltalk-80的发布引起了人们广泛的关注，导致了在80年代早期到中期其它面向对象语言的蓬勃发展，有的是对传统语言的扩充，有的是新开发的面向对象语言，其中最有代表性的包括Objective-C(1986年)，C++(1986年)，Self(1987年)，Eiffel(1987年)，CLOS(1986年)，以及Object oriented Pascal等等。面向对象的应用也被广泛地扩大了。1986年，Grady Booch首先提出了“面向对象设计”的概念，从那以后，越来越多的人投入到面向对象的研究领域。一方面，面向对象方法向软件开发的前期阶段发展，包括面向对象分析、面向对象设计；另一方面，面向对象技术在越来越广泛的软硬件领域得以发展，如面向对象数据库、面

向对象操作系统、面向对象软件开发环境、面向对象的智能程序设计、面向对象的计算机体系结构等等。

人们预计在90年代，面向对象技术将会在更深、更广、更高的方向上取得进展：

① 更深的方向，如面向对象技术的理论基础和形式化描述、用面向对象的概念设计操作系统等；

② 更广的方向，如面向对象的知识表示、面向对象的仿真系统、面向对象的多媒体系统等；

③ 更高的方向，如从思维科学的高度来丰富面向对象方法学的本质属性，突破现有的面向对象技术的一些局限，研究统一的面向对象范型(Paradigm)。

6.1.2 一些基本概念

在继续深入讨论面向对象方法之前，让我们首先了解一些基本概念。

(1) 对象

对象一词，应用广泛，场合不同，含义各异。一般来说，客观世界中任何有确定边界、可触摸、可感知的事物，包括概念(如速度、时间等)均可看作对象。任何事物，均有其各自的属性和行为。当考察其某些属性和行为并进行研究时，它才成为对我们有意义的对象。因此，在系统分析和系统构造中，对象是对客观世界事物的一种抽象，是由数据(属性)及其上操作(行为)组成的封装体。

对象具有如下主要特点：

① 自治性　对象的自治性是指对象具有一定的独立计算能力。即对于给定的输入，经过状态转换，对象能产生输出。其中，对象自身的状态变化是不直接受外界干预的，外界只有通过发送消息对它产生影响，从这个意义上说，对象具有自治性。

② 封闭性　对象的封闭性是指对象具有信息隐蔽的能力。具体地说，外界不能直接改变对象的状态，只能通过向该对象发送消息来对它施加作用。对象隐蔽了其中的数据及操作的实现方法，对外可见的只是该对象所提供的接口——操作。

③ 通信性　对象的通信性是指对象具有与其它对象通信的能力，即对象能够接收其它对象发来的消息，也能向其它对象发送消息。通信性反映了不同对象间的联系，通过这种联系，若干对象可协同完成某项任务。

上述特点分别刻画了对象的不同方向性质。自治性反映了对象独立计算的能力，封闭性和通信性说明对象是一个既封闭又开放的相对独立体。

(2) 类

具有相同属性和服务的对象的集合。类作为模板，为属于该类的所有对象提供了相同的结构、相同的操作(集)、对其它对象具有相同的关系和相同的语义。对象是类的实例。

(3) 属性

每一对象的属性是一些有着确定值的、用于描述对象状态信息的数据。

(4) 服务

为了完成某一任务，一个对象所提供的、并体现其责任的操作。属于同一类的所有对象共享相同的服务。

(5) 消息

一个对象为实现其责任而与其它对象的通信。在面向对象方法中，对象之间只能通过消息进行通信。

(6) 继承

表达类之间相似性的一种机制，即在已有的类的基础之上增量构造新的类，前者称为父类(或超类)，后者称为子类。子类除自动拥有父类的全部属性和服务外，还可以进一步定义新的属性和服务。如果子类只从一个父类继承，则称为单继承；如果子类从一个以上父类继承，则称为多继承。

6.1.3 同结构化方法的比较

结构化方法强调过程抽象和模块化，将现实世界映射为数据流和加工，加工之间通过数据流进行通信，数据作为被动的实体被主动的操作所加工，是以过程(或操作)为中心来构造系统和设计程序的。

面向对象方法把世界看成是独立对象的集合，对象将数据和操作封装在一起，提供有限的外部接口，其内部的实现细节、数据结构及对它们的操作是外部不可见的，对象之间通过消息相互通信，当一个对象为完成其功能需要请求另一个对象的服务时，前者就向后者发出一条消息，后者在接收到这条消息后，识别该消息并按照自身的适当方式予以响应。

面向对象方法和结构化方法相比，具有以下一些特点：

(1) 面向对象方法强调把问题域的概念直接映射到对象以及对象之间的接口，符合人们通常的思维方式，减少了结构化方法从问题域到分析阶段的映射误差；

(2) 面向对象方法从分析到设计再到编码采用一致的模型表示，后一阶段可以直接复用前一阶段的工作成果，弥合了结构化方法从数据流图到模块结构图转换的鸿沟，减少了工作量和映射误差；

(3) 在客观世界以及作为它的映射的软件系统中，实体的结构是相对稳定的。面向对象方法通过把属性和服务封装在"对象"中，当外部功能发生变化时，保持了对象结构的相对稳定，使改动局限于一个对象的内部，减少了改动所引起的系统波动效应。所以，按照面向对象方法开发的软件，具有易于扩充、修改和维护的特性；

(4) 面向对象方法具有的继承性和封装性支持软件复用，并易于扩充，能较好地适应复杂大系统不断发展和变化的要求。

目前，对面向对象方法的研究方兴未艾，已经提出许多种不同的面向对象的分析、设计方法，还没有形成一个统一的规范。比较典型的方法有：Coad-Yourdon 方法、Booch 方法、Shlaer-Mellor 方法、Rumbaugh 等五人提出的 OMT(Object Modeling Technology)方法、Jacobson 提出的 use case 驱动的方法等。这些方法从本质上来说没有根本区别，都是以对象为中心来构造模型、组织系统，只是各种方法的侧重点、符号表示和实施策略上有所不同。下面我们主要以 Coad-Yourdon 方法为例介绍面向对象的软件开发过程。

Coad-Yourdon 方法认为，人类在认识和理解现实世界的过程中，普遍运用着下面三个构造法则：

① 区分对象及其属性。例如，区分一棵树和树的大小或空间位置关系；

② 区分整体对象及其组成部分。例如，区分一棵树和树枝；

③ 不同对象类的形成及区分。

Coad-Yourdon 方法就是基于这三个常用的构造法则。分析阶段由五个主要活动组成：标识类及对象、标识结构、标识主体、定义属性及实例连接、定义服务及消息连接。但这并不意味着这五个步骤必须顺序进行。事实上，在实际的工作中，这五个步骤经常是根据需要交叉进行的。在下面的内容里，我们就具体介绍这五个步骤。

6.2 标识类及对象

6.2.1 为什么要标识类及对象

类及对象意为“一个类以及属于这个类的所有对象”。

标识类及对象的主要目的是为了使一个系统的技术表示更紧密地切合现实世界的概念视图。对问题域中类及对象的抽象影响着可理解性及有效的通信，在完整理解一个问题域之前就开始写需求规格说明是没有任何意义的。

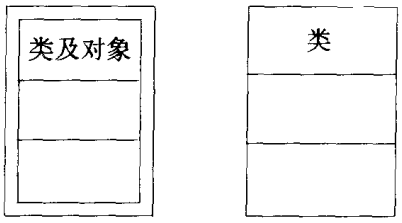
强调类及对象的另一个动机是希望能建立一个稳定的框架模型以进行分析和规约。例如，今天的空运控制系统中的类及对象可能同五年前的空运控制系统中的类及对象没有多大区别——但是那些类及对象的属性和服务却可能发生了巨大的变化。随着时间的推移，类及对象是相对稳定的，因此提供了复用分析结果的基础。

标识类及对象的最后一个目的是为了避免从系统分析向设计过渡时改变系统的基本表示。在 70 年代和 80 年代，分析和设计之间的鸿沟似乎是不可逾越的。多年来在分析中使用的是网络结构(数据流图)，而在设计中使用的则是层次结构(模块结构图)，这一状况使开发者感到十分不便，无法跟踪开发过程。而这两点恰恰对于大型、关键系统的开发而言是十分重要的。表示法的不同是引起分析/设计混乱的根本原因所在，需要说明的是，应用面向对象分析和面向对象设计并不意味着必须使用面向对象的程序设计语言，但是实现阶段如果使用面向对象的程序设计语言将会有所帮助。

6.2.2 如何表示类及对象

1. 符号表示

类及对象的符号代表一个类及它的所有对象，如图 6.1(a)所示。



(a) (b)

图6.1 面向对象分析中的“类及对象”符号

其中内层矩形表示类，它被分成三个部分，依次标明类名、属性名和服务名；外层矩形表示该类的对象。这就引出另一个符号，当一个类没有任何属于它的对象时(即抽象类)，就用 6.1(b)所示的符号表示。

2. 从哪儿入手

首先应该考察并研究问题域本身。用户的需求总是有其应用背景的，只有充分理解了问题域，才能真正明白用户的需求及未来系统在问题域中所处的位置、确定系统元素、划定系统边界以及同问题域中其它元素之间的关系。研究问题域可以通过亲身实践、同用户交流、请教问题域专家或从专业书籍中找到有关该问题域的权威描述。

其次，通过不断地和用户交流，包括座谈、调研、正式的需求文档提交等手段，充分理解用户的需求，从中捕获同问题域和系统责任密切相关的类及对象，不断地作原型，请用户参加评审，听取用户的反馈意见，修正我们对用户需求的理解。

最后，尽可能多地收集相同或相关问题域中其它系统的信息，以及任何可以得到的关于问题域的材料，包括文字说明和图表，同样有助于确定类及对象。

3. 寻找什么

为了确定潜在的类及对象，请寻找：问题域的结构、相关系统、设备、需要记忆的事件、发挥的作用、地点和组织单元。以下所列各项是按照最有可能发现类及对象的优先级从高到低的顺序给出的。

(1) 问题域结构

问题域中的结构对于发现类及对象和表示问题域的复杂性具有重要的意义，因而它本身也是OOA方法中的五个活动之一。其中一般/特殊结构和整体/部分结构是最为重要的两种结构，我们将在下一节中详细讨论。

(2) 相关系统

正在考虑的系统将与哪些相关系统和“外部边界”发生相互作用？这种相互作用可能是通过硬件连接的(比如用电缆连接的系统)、相互传递信息的(比如一架自动报告其高度的飞机)或人机交互的结果(比如用户描述的车辆)。

(3) 设备

正在考虑的系统需要同哪些设备相互作用?有些设备可能与本系统交换数据和控制信息，但要注意的是不要从实现的角度而应从用户需求的角度来考虑是否增加类及对象，把有关实现的细节推迟到设计阶段考虑，以便在具体实现发生变化时，避免大量的重复性工作。

(4) 需要记忆的事件

下一步，考虑系统是否有必要观测和记录时间点或历史事件。例如，在某个时间点上有人领取了机动车牌照，这是一个合法的事件，系统必须记录在案；又如，如果系统正在监测一个核反应堆，当发生故障时，关于整个历史事件的信息必须都记录下来：谁、什么、何时、何地、如何、为什么等。

(5) 发挥的作用

在所考虑的系统中，人将发挥什么样的作用，或扮演什么样的角色?通常，人在系统中扮演两种类型的角色：系统用户(即使用该系统的人员)；不与系统直接打交道，但系统处理的却是有关他们的信息(如银行的客户)。这两类人在系统中扮演的角色不同，我们在后面讨论属性和服务时，会进一步分析二者之间的区别。

(6) 操作过程

系统需要保持什么操作过程，以指导用户和计算机的交互?当系统责任包括对特定操作或工作流程步骤的记录时，就需要标识这样的类及对象。其中的属性可能包括操作过程名、授权

级别以及对过程的描述。

(7) 地点

所考虑的系统需要了解哪些物理地点，办公室或场所？例如，一个嵌入式系统将被安置在特定经度、纬度、高度和地形的地点。

(8) 组织单元

与系统有关的人员属于哪一个组织？例如，某个职员在某银行工作，那么“银行”就可能成为一个潜在的类及对象。进一步，如果系统需要记录该银行的信息(如名称、经理、地址等)或提供与该银行有关的处理(如某一天的资金流通情况)，就需要“银行”这个类及对象。

总之，通过考察问题域结构、相关系统、设备、需要记忆的事件、发挥的作用、地点和组织单元等方面，寻找潜在的类及对象。一旦找到一个候选的类及对象，请参照下一段的内容进一步核查。

4. 考虑和挑剔什么

通过上面的步骤，你可能已经发现了一些候选的类及对象，但是否应该把它们作为最终模型中的类及对象呢？这时应该考虑以下的问题：需要的记忆、需要的行为、(通常)多于一个属性、(通常)类中多于一个对象、公共属性、公共服务、基于领域的需求、可推导的结果。

(1) 需要的记忆

系统需要记忆属于某个类的对象的任何信息吗？请检查问题域和系统责任，能否描述属于某个类的对象？该对象的一些潜在的属性是什么(例如，一个职员的潜在属性包括姓名、密码和权限)？现实世界中对象的信息是否是正在考虑的系统所关心的？系统是否需要记忆与该对象有关的情况？如果不是，则该类及对象存在的合法性和必要性就值得怀疑了。

记住：现实世界中存在着许多有趣的类及对象，也可能在与客户的讨论中经常出现，但是它们并不一定都与系统有关。

(2) 需要的行为

对象需要提供某些行为(或处理)吗？只要需要的记忆适用，那么服务也将需要——至少需要创建、连接、访问和释放该对象。

然而，一个对象或许需要有一些服务，但并不需要记住某些信息(属性)。例如，一个系统外的类，该类只有一个对象实例，就可能只有服务(而没有属性)来处理相应的系统接口。

(3) (通常)多于一个属性

当分析员考虑问题的层次过低时，这条准则有利于过滤掉一些可能的类及对象，如果一个对象(比如“安装位置”)仅有一个属性(比如“地址”)，则该对象存在的必要性就值得怀疑了，也许“地址”作为一个属性在若干个类及对象中出现比它自己作为独立的对象存在更好。关键是细节之间的调整，类及对象由属性描述，而属性则进一步在类及对象说明中描述。

(4) (通常)类中多于一个对象

在类及对象的符号中标上“这个车辆”或“那个车辆”是值得怀疑的。如果只有单个对象的类确实反映了问题域，那么它的存在是合理的，比如一个空运控制系统可能有“雷达”这个类，而在问题空间中确实只有一个雷达，则雷达作为一个类可能是合适的。

假如在组装结构的根部有一个“主管”对象，但是如果存在另一个有相同属性和服务的类及对象，并且它也确切地反映了问题域的现实，那么就考虑使用同一个类及对象。而如果存在另一个有相似属性和服务的类及对象，并且它也刻画了现实世界，那么考虑使用一般/特殊结

构，我们将在下一节详细讨论。

(5) 公共属性

你能确认一组属性适合于一个类的每个对象吗?系统每次询问一个对象时，它的每一个属性都应该有确定的值。例如，在一个不动产管理系统中的类及对象“建筑物”可能包括地点、价格、标价时间、面积等属性，还有房间数目和浴室数目。如果有些属性只适用于某种类型的建筑物，那么就考虑使用一般/特殊结构，我们将在下一节详细讨论。

(6) 公共服务

你能标识一组公共服务——即适合于一个类的每个对象的行为(或处理)吗? 服务可能在算法上比较简单(创建、连接、访问、释放)或在算法上非常复杂(计算、初始化/监控/终止)，如果类中每个对象的服务是相同的，那就好办了。然而，如果服务因不同的对象而异，就说明需要增加一个一般/特殊结构。

以上讨论表明在研究一个问题域中的初始对象集时，应从一般性的角度研究公共属性和服务，以后当你更加详细地考虑属性和服务时，你将发现可以加进模型中更多的详情以及区别。

(7) 基于领域的需求

基于领域的需求是在不考虑用于构造(设计和实现)系统的计算机技术时，系统必须有的需求。当雷达和传感器（如测量温度、压力或动力的设备)出现在问题域中时，不管最终将采用什么样的计算机技术，我们都将在模型中见到对应于“雷达”和“传感器”的类及对象符号。而系统体系结构(集中式、分布式或复制式)、磁盘驱动器、显示终端，为了较高的吞吐率而做的批处理，时间和空间的折衷等等都是设计和实现阶段的考虑；该项目是选用袖珍显示设备、膝上计算机、复杂的图形系统还是大屏幕显示器也是设计上的选择。作为一个分析员，应首先集中精力于确定系统需要什么信息和行为，使用原型定义和精化所需要的信息内容究竟是什么。但是，一些关于设计的必要约束是应当作为需求提出来的，如电梯控制系统必须采用某种调度算法等。

(8) 可推导的结果

最后，应避免在模型中包括可以推导出的结果，例如在一个已经有客户出生日期的系统中再包含“客户的年龄”，这样会造成信息冗余，容易导致不一致性，给分析模型带来不必要的麻烦。出于效率的提高，可以暂存某些计算的结果，这应在设计和实现阶段加以考虑。分析阶段的需求大多数都是简明地陈述为执行计算的能力。

应当指出，标识对象是 OOA 的一件十分关键的任务，同时又强烈地依赖于具体的问题域空间和分析人员的经验和才智。目前，还没有一个通用的“放之四海而皆准”的标准认定规则。上面叙述的原则也只是来自某些专家的成功经验，既不是标准也不可能涵盖一切。因此，如果读者在自己的开发实践中发现其它一些原则，也是很正常的。

6.3 标识结构

结构是一种思维组织的方式，用来反映问题域空间事物之间的复杂关系。我们这里讨论的结构包括两种：一般/特殊结构和整体/部分结构。一般/特殊结构针对的是事物类之间的组织关系，体现了现实世界中事物的一般性与特殊性。例如，交通工具同汽车、飞机、轮船之间形成

了一般/特殊结构，将汽车、飞机、轮船这几种事物的共有特性概括在交通工具之中，而汽车、飞机、轮船等专有的特性（包括属性和服务）则包含在各自的类中。整体/部分结构表示事物的整体与部分之间的组合关系，例如，把汽车看成一个整体，那么发动机、变速箱、刹车装置等都是汽车的部件，相对于这个整体，就分别是一个局部。

6.3.1 为什么要标识结构

一般/特殊结构和整体/部分结构能使分析人员和领域专家集中精力在具有多个类及对象的复杂问题上，而且，使用结构能使分析人员考虑到系统责任的边界，并揭示那些尚未发现的类及对象。除此之外，一般/特殊结构表达的继承性，使得通用的属性和服务一旦被标识，即可被特殊类直接继承。

6.3.2 如何标识一般/特殊结构

1. 表示法

一般/特殊结构的符号如图 6.2 所示。

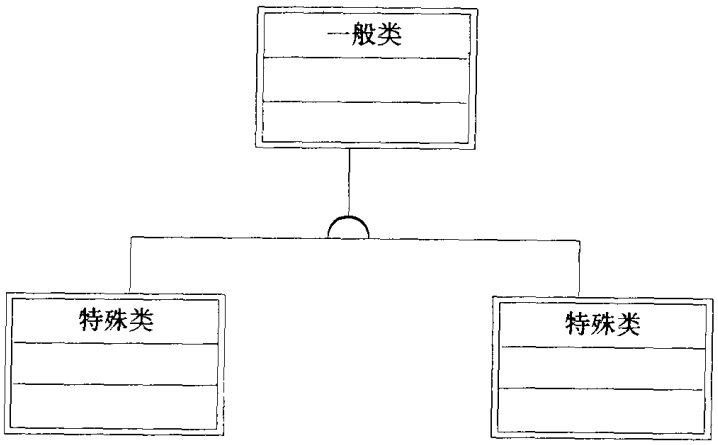


图6.2 一般/特殊结构的记号

上述符号的顶部是一般类，下部是特殊类，它们之间用线段连接。半圆形的标记表明它们之间形成了一般/特殊结构，从半圆弧的顶部引出的一条线连接的是一般类，其它为特殊类。通常，一般类总是放在上部，而特殊类放在下部，这样的布局有助于模型的理解。一般/特殊关系的连线端点位置（连到表示类符号的内部矩形的边上）表明这是类（而不是对象）之间的关系。

每个特殊类的名字必须能够充分地反映它自己的特征。比较合适的特殊类的名字通常由相应的一般类的名字加上能描述该特殊类系性质的限定词来组成，例如，对于名为 Sensor（传感器）的一般类，其特殊类可称为 Standard Sensor（标准传感器）或 Critical Sensor（临界传感器），而不称之为 Critical（临界）。

所有最底层的特殊类必须使用类及对象符号，而其它地方则既可用类及对象符号也可只用类符号。

2. 实施策略

将每个类看成是一般类，针对它的每个潜在的特殊类考虑以下问题：

- 它属于该问题域吗？

· 它在系统责任范围内吗?

· 存在继承性吗?

· 特殊类满足上一节中评价类及对象的准则吗?

以类似的方式,将每个类看成是特殊类,并对它的潜在特殊类考虑上述同样的问题。

检查以前相同和相似问题域的面向对象分析结果,寻找可以直接重用的一般/特殊结构并吸取有关教训;另外,如果存在多个特殊类,则首先考虑最简单的和最复杂的特殊类,然后处理其它的。例如,"洗衣机"作为一般类就存在很多的特殊类,从早期最简单的单缸洗衣机,发展到全自动模糊感应洗衣机,依此建立的模式还可以发现许多功能和复杂性介于二者之间的中间产品。

再如,把"飞机"类考虑成一般的,它可以按下述不同的分类方式进行特殊化:

· 喷气式飞机和直升机;

· 民用飞机和军用飞机;

· 固定机翼飞机和活动机翼飞机;

· 商用飞机和私人飞机。

让我们来考察喷气式飞机和直升机作为特殊类的情况:

· 喷气式飞机和直升机作为特殊类在所考虑的问题域内有区分的必要吗?如果在该问题域内无需这种区分,则不需要特殊化;

· 系统需要区分喷气式飞机和直升机吗?需要为它们之间的不同保存什么样的信息?如果不需区分,则不需要特殊化;

· 存在继承性吗?即一些属性和服务适用于所有的飞机,还有一些属性和服务只适用于喷气式飞机,另外一些属性和服务只适用于直升机;

· 这些特殊类满足上一节中评价类及对象的准则吗?

作为例子,让我们考察此时可能会遇到的一些变异情况。如果喷气式飞机和直升机只在飞机类型上有区别,只需在飞机类中包含一个描述"飞机类型"的属性,而无需增加一般/特殊结构;如果特殊类之间的唯一区别是系统需要了解喷气式飞机的最大推力(而无需了解直升机的最大推力),那么只需在飞机类中包含一个描述"最大推力"的属性,而无需增加一般/特殊结构;如果某个特殊类没有增加属性和服务,它就不是必需的——除非它同时具有多个直接的父类。

同样,对于每个类,把它看成是特殊类,针对前面提出的问题,考察它的潜在一般类。

评价一般/特殊结构的一个准则是看它是否真正反映了问题域中的一般性和特殊性,不要为了抽取几个公共属性而人为引入一般/特殊结构,如图 6.3 所示,模型的可理解性远比分解出一些公共属性更为重要。

3. *层次和网络*

每个一般/特殊结构均形成层次或网络。在实践中,一般/特殊结构的最普通形式就是层次,如图 6.4 所示。

在本例中,一个 Person(人)可以是 Owner(所有者),Clerk(职员)或 Clerk 兼 Owner,在这样的层次结构中,特殊类之间存在着相当的冗余信息。

借助于一般/特殊网络结构可以减少上述的冗余信息,并有助于发现更多的问题域中的特殊类,而且可以显式地表示更多的属性和服务的共性。

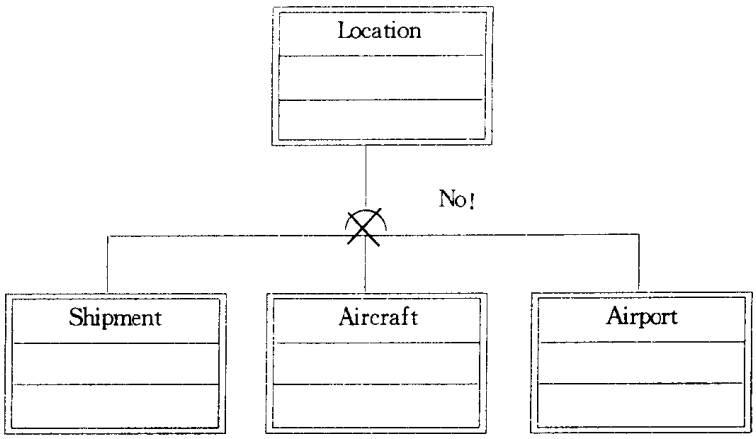


图6.3　一个不适合的一般/特殊结构

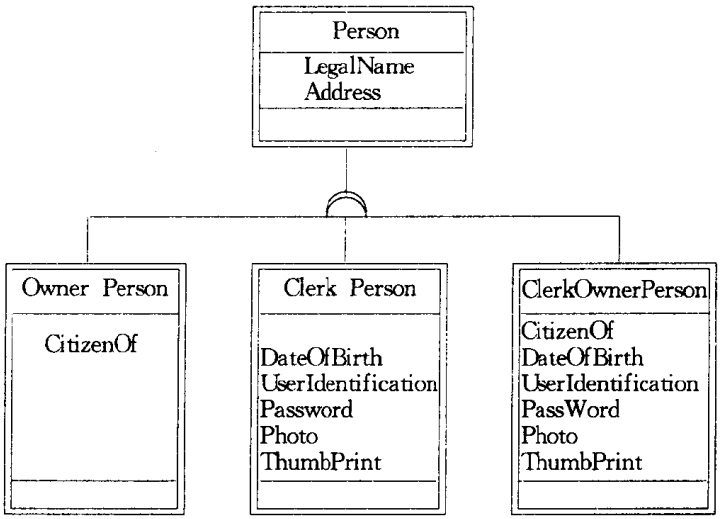


图6.4　层次的一般/特殊结构

在图 6.5 所示的例子中，问题域中包含“人”这个类，任何人都属于 Owner（所有者）、Clerk（职员）或身兼两者。特殊类 Owner Clerk Person 具有多个直接的一般类，这意味着该类的对象继承了来自多个祖先的属性和服务，而且如果这个类本身还定义了新的属性和服务，则它们也将出现在该类的对象中。

因此，网络结构能够

- 更加强调一般/特殊关系；
- 显式地捕捉共性；
- 对模型的复杂程度影响较小。

然而，随着特殊类的不断增加，网络结构将变得非常复杂。如果出现这种情况，可以考虑将网络的某一部分重新组合成另一个层次。同时要注意解决在一个以上的直接一般类中使用相同的属性名和服务名时造成的命名冲突问题。

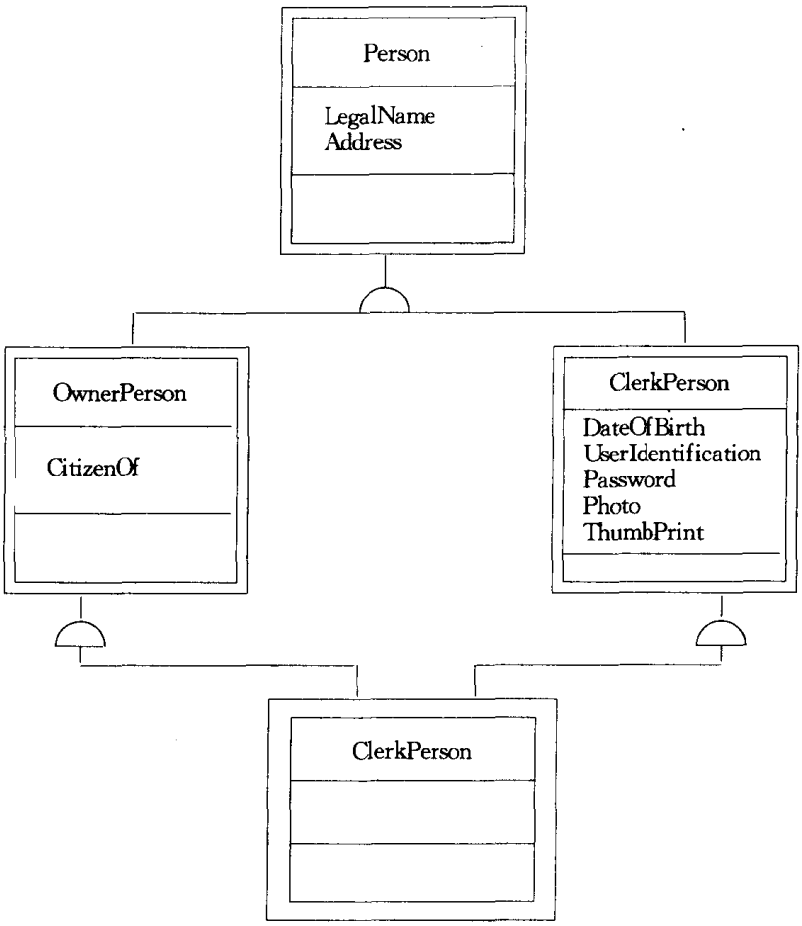


图6.5　网络的一般/特殊结构

6.3.3　如何标识整体/部分结构

1. *表示法*

整体/部分结构的符号如图 6.6 所示，顶部是整体对象，下面是部分对象，它们之间用线段连接。从三角形标记的顶点引出的垂直于对边的线段连接的是整体对象，其它为部分对象。一个整体对象可以有不同类型的部分对象，一个部分对象也可以同时属于多个整体对象，可以在连接整体对象和部分对象的线段上标出数字或区域，表明整体对象和部分对象在数量上的约束关系。整体/部分结构的连线端点位置（连到表示对象符号的外部矩形的边上）表明这是对象（而不是类）之间的映射关系。

在图 6.7 所示的例子中，飞机是一个整体对象，它可能

- 没有发动机；
- 最多有四台发动机。

而“发动机”作为部分对象，它可能

- 不是飞机的一部分；
- 至多是一架飞机的一部分。

2. *实施策略*

首先确定潜在的整体/部分结构，考虑以下一些变化的情况：

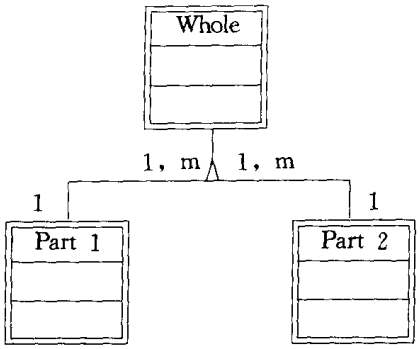


图6.6　整体/部分结构表示法

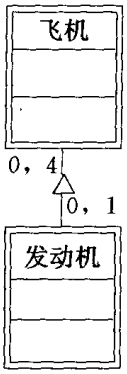
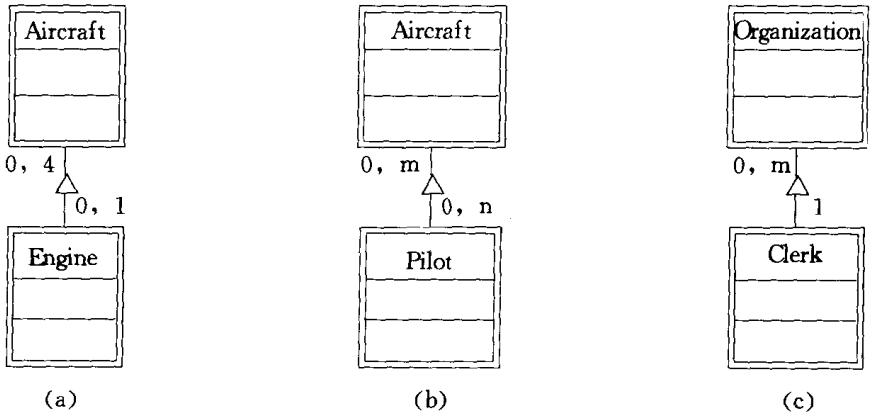


图6.7　整体/部分结构实例

- 组装——部分；
- 容器——内容；
- 集合——成员。

除此之外，还应查看以前相同和相似问题域的面向对象分析结果，确定能直接重用的整体/部分结构，并吸取有关教训。

让我们还是通过一些具体的例子来考查以上的变化情况：



(a)组装/部分　　(b)容器/内容　　(b)集合/成员

图6.8　一些整体/部分结构的例子

在确定了潜在的整体/部分结构以后，将每个对象看作整体对象，对它的潜在部分对象，考

虑以下问题：

①是否属于问题域？

②是否属于系统责任？

③是否只包含了整体对象内部的一个属性或状态？

以同样的方式，将每个对象看作部分对象，对它的潜在整体对象，考虑以上类似的问题。

例如，考虑一架飞机，发动机作为一个潜在的部分对象，作如下的考查。

④发动机在该问题域内有意义吗？如果问题域是“就餐服务”，也许就没意义了；如果问题域是“空中运输”，就可能有意义了。此处我们考查的问题域是“空中运输”，回答：是的，发动机在问题域内。

⑤发动机属于系统责任吗？系统需要知道并与发动机进行交互吗？回答：是的，对“空中运输”而言，发动机属于系统责任。

⑥发动机对象是否只包含了整体对象的一个属性？如果系统责任只包括飞机状态（工作与否）或发动机状态（工作与否），那么也许不必引入“发动机”作为类及对象；相反，在飞机类中定义一个“操作状态”属性，就可以用简单的模型捕获系统责任。然而，如果关于发动机的系统责任不只包括状态，例如，还包括生产厂家、型号、序列号、出厂日期、检修日期等属性。那么，回答：是的，发动机作为类及对象是需要的。

⑦是否为处理问题域提供了有价值的抽象？回答：是的。

通过对一般/特殊结构和整体/部分结构的刻画，可以将问题空间中出现的某些事物有机地联系起来，合理地纳入到范围更广的事物群中，有助于人们更好地理解和把握未来的系统。

6.4 标识主题

主题是一种控制复杂性的机制，用于指导读者或用户研究大型的复杂模型。标识主题的主要原因是以一般/特殊结构和整体/部分结构为代表的问题域的复杂性，主题所依据的原理是整体/部分关系的扩充，在这种方式下，主题就是用来同整个问题域和系统责任这个“总体”进行通信的“部分”。

6.4.1 为什么要标识主题

一个面向对象分析模型中类的数目依赖于问题域和系统责任的广度和深度，包含35个类的系统是中等规模的；包含110个类的系统是大规模的；至于包含多个子问题域的复杂问题，如空运控制，也许有五个子问题域，其中每个包含50—100个类。

对任何一种方法及其应用来说，其成功的关键因素之一是它能提供便利通信的能力，从而避免分析员和客户之间信息过载。因此对于较大规模的模型而言，问题的关键是如何指导读者理解模型的不同部分。

George Miller 的著名文章《神奇的数字7，加减2：我们处理信息的能力的某种限制》(Psychological Review，March 1956)大大影响着人们为在软件工程领域提供抽象所作的种种努力。Miller 指出，人们在一个时间内的短期记忆似乎限制在5—9件事情之内(除非这个人学会了使用联想记忆法的技巧了)。

依据 Miller 的研究成果，面向对象分析模型中增加了一个主题层，通过对模型进行划分，

指导用户在观察一个模型时，注意力集中在模型的不同部分，从而减轻了用户理解上的负担。同时，主题层提供了从更高层次来观察模型全貌的手段，有助于读者复审模型并简明地概括所考虑的问题空间之内的主题。

6.4.2 如何标识主题

1. 如何选择主题(包括一般/特殊结构和整体/部分结构)

为每个结构增加一个主题，为每个不属于任何结构的类及对象增加一个主题。检查以前相同和相似问题域的面向对象分析结果，寻找可以直接重用的主题并吸取有关教训。

2. 如何精炼主题

为了精炼主题，可从问题域和接口复杂性两方面入手：

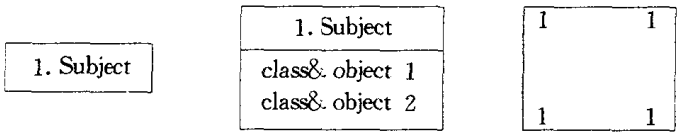
①使用问题域来精炼主题，即使用整体/部分结构对问题域进行划分，而不是按功能分解的方法进行划分；

②按照高内聚、低耦合的原则，通过使主题之间依赖性和交互性达到最小来确定主题：依赖性由结构和实例连接(将在6.5节讨论)表示，而交互性则是由消息连接来表示的(将在6.6节讨论)。通过使用结构、属性和服务层的信息可以帮助有效地确定主题；

③如果主题的数目超过7个左右，则需进一步精炼主题，从而形成不同层次的主题。高层次的主题可以包含低层次的主题，越高层次的主题抽象级别越高，随着主题层次的下降，不断增加细节，从而指导用户更好地理解模型。

3. 如何构造主题

为每个主题命名并编号，主题有三种表现形式：萎缩方式、半展开方式和全展开方式。如图6.9所示，萎缩方式图示为简单的矩形框，内部有主题名称和编号；半展开方式图示为分成上下两部分的矩形框，上部是主题名称和编号，下部列出本主题包含的类及对象名和其它主题名；全展开方式是在显示面向对象分析的其它层时，主题以边框的形式给出，所有属于该主题的类及对象和其它主题都在其中，在主题边框的各个角的内部标上主题标号。为了更好地引导读者，一个类及对象可以属于不只一个主题。



(a) 萎缩方式　(b) 半展开方式　(c) 全展开方式

图6.9 主题表示符号

4. 何时引入主题

那么什么时候在模型中引入主题呢？这依赖于模型自身的复杂性。对于一个很小的系统来说，模型本身足够简单，也许根本就不需要主题；对于一个中等规模的系统，应该首先识别类及对象和结构，然后引入主题，并将它作为指导读者理解整个模型的手段；然而针对很大的项目，需要首先标识主题，对问题域进行划分，并分配给不同的任务组。在这种情况下，作为高级分析人员应该快速地标识类及对象和结构，从而对主题进行初步标识。随着对问题域和系统责任理解的不断加深，再对主题进行精炼和调整。

6.5 定义属性

属性是用来描述对象状态的数据，在类的每个对象中均有确定的值。属性为类及对象提供了更多的描述细节，明确了一个类的名字究竟在问题域和系统责任中意味着什么。

6.5.1 为什么要定义属性

主要原因是属性能为类及对象和结构提供更多的细节。

属性涉及到分析和选择，例如在一个人事管理系统中，当选择“人”的属性时，确定属性涉及到分析和选择，作为一个人他(她)有很多属性，身高、体重、相貌也许在有些场合有用，甚至可能是重要的决定因素，但在一个人事管理系统中，强调的却是年龄、学习经历、工作经历等等。针对不同的问题域和系统责任，类及对象通过属性为系统保存的信息也不同。

属性描述隐蔽在对象内部的信息，由该对象的服务专门操纵。我们将属性及操作这些属性的专有服务看成是一个不可分割的整体。如果系统的其它部分需要访问或操作某个对象内部的属性，它必须通过发出一条同该对象中定义的某个服务相对应的消息来完成。所以，随着面向对象分析方法的深入，信息隐蔽和数据抽象就逐渐起作用了。

值得注意的是，随着时间的推移，问题域中的类将保持相当的稳定性，而属性却比较容易改变。例如，考虑空运控制系统中的“飞机”类，现在的飞机只传递识别标志和高度两个参数，而几年后的飞机可能要报告更多的参数，如爬高/下降率、机翼位置、机上各个子系统的状态等。据此地面系统就可以通过机翼的位置判断飞机何时在转弯，而不是像现在这么仅仅靠雷达来推断。“飞机”这个类仍然保持不变，但属性的数量以及属性上的专用服务的复杂性却发生了变化。

6.5.2 如何定义属性

定义属性可以分为以下几个步骤：

- 标识属性
- 定位属性
- 标识实例连接
- 检查特例
- 指定属性

属性放在类及对象或类表示符号的中间部分，如图 6.10 所示。

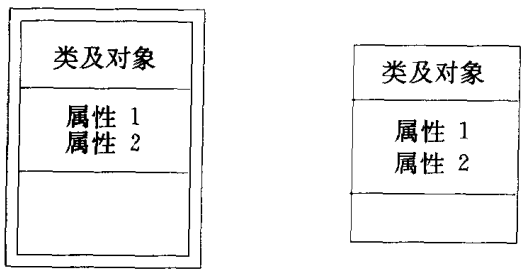


图6.10 属性的表示符号

1. 标识属性

作为系统中存在的某个类的对象，应该为系统责任保持一定的信息，这正是对象中属性的作用，从单个对象的角度考虑如下问题：

①一般情况下怎样描述该对象？

②在本问题域中怎样描述该对象？

③ 在本系统责任内怎样描述该对象？

④ 该对象需要知道什么？

⑤ 该对象需要记住什么状态信息？

⑥ 该对象能处于什么状态？

检查以前相同和相似问题域的面向对象分析结果，确定哪些属性能够直接重用？在定义属性方面可以吸取什么经验和教训？

例如，对于与所考虑系统有关的人，系统究竟应该保存他（她）的哪些属性呢？对于称作“主管”的类及对象，系统应该知道“主管”的哪些具体情况呢？有些属性是相当直接的，如姓名、地址、头衔等，但还会有更多其它的属性。从下面的角度考虑这个问题。

（1）原子概念

使每个属性表示一个“原子概念”，即单个值或紧密相关的一组值。这个概念可能是一个数字（如驾驶执照号码），或由数字和字符串构成的一组值（如出生年月日、地址）。表示“原子概念”的动机是为了产生一个供人观察的简单模型，对象中包含不多的几个属性名字以及数据集合以便于理解。

（2）延迟到设计——规范化

引入新表消除数据冗余（规范化）并获得可以接受的性能，这两者之间的折衷要推迟到设计阶段。在设计阶段，规范化要求每个数据项没有内部结构（这个规则属于第一范式的内容）。但是，分析员在建立初始模型时并不关心规范化问题。实际上，在这个早期阶段，没有办法知道设计人员是否会对数据规范化以及在哪个层次上进行规范化。

（3）延迟到设计——标识机制

另一个需要延迟到设计阶段的问题是如何选择实际使用的标识机制，如关键词、相关表、指针等等。

属性标识提供了一种引用对象及其对象连接的便利手段。每个对象都需要这样的一个标识符，所以作为一种约定，为使模型图保持简单，每个对象都有一个隐含的标识符（id）和连接标识符（cid）。之所以说这些标识符是隐含的，是因为它们并没有明显地出现在模型的属性层中。

定义这些标识符的主要原因是为了便于描述对象服务，即为了简化一个对象或对象之间连接的描述；另一个原因是为了避免使用现实世界的标识符作为唯一标识符。唯一标识符必须是唯一的，而且一旦放入系统中就不再改变，这种唯一性意味着必须作出某种设计上的抉择，因为现实世界的标识符不能绝对保证唯一性。例如，在一个人口管理系统中，身份证号似乎是唯一的，可以用身份证号代替 id 作为对象标识符。但现实中的标识符是会出现重复的，某个职员可能会输错一个身份证号码；一段时间以后，当另一个职员在试图输入一个合法的身份证号码时却出现了与输错的身份证号码重复的号码。

因此，在分析工作中不要使用现实世界的标识符及其套接字，以免使模型复杂化，并带来

本应在设计阶段定义的可能会发生设计错误的细节。应该使用隐含标识符(id 和 cid)作为引用对象的“句柄”,在设计时再选择是用多属性的关键词(现实世界的标识符加上一个套接字),还是用能够保证唯一的单属性关键词(系统本身产生的)。

(4) 延迟到设计——保持一个可导出的属性

还有一个需要延迟到设计阶段考虑的问题就是,是否要保持一个可以推导出的属性。这通常是CPU 时间和储存约束之间的一个折衷。对面向对象分析而言,一般只指定计算需要的服务,而不指定相应的可导出的属性。

2. 定位属性

前面我们已经标出了属性,接着就要把每个属性放到它最合适的类及对象之中。定位属性要注意两个方面,一方面必须是对现实世界的准确映射,以达到最大的稳定性和建模的一致性;另一方面,在一般/特殊结构中,放在结构最上层的属性将适合于所有的特殊类。因此,如果某个属性适合于某一层的所有特殊类,则应将它上移到相应的一般类。如果发现某个属性的值有时有意义,有时却不适用于某些对象,则应回顾一般/特殊结构的策略,看是否存在着另一个一般/特殊结构。

3. 标识实例连接

属性描述的是对象状态,实例连接则通过刻画一个对象与其它对象的映射关系,进一步加强了这种描述能力。具体地说,实例连接是问题域映射模型,该模型反映了某个对象对其它对象的需求,以完成其系统责任。实例连接用两个对象之间的连线来表示,如图 6.11。该线的端点位置表明实例连接是对象之间而不是类之间的一种映射关系。每条实例连接线上均标有数字或者范围,反映了该对象对其它对象的约束,该数字或范围表明可能发生的映射数目或范围,可以直接给出上下限,如“1,5”,当映射约束是固定数目时,也可以使用一个数字表示。例如,一个特定的飞行计划只能是专为一架飞机而制订的,而任何一架飞机可以有零到多个飞行计划,这就是说,一个飞行计划只能有一个实例连接到某架飞机,而一架飞机可以有零到多个实例连接到飞行计划。

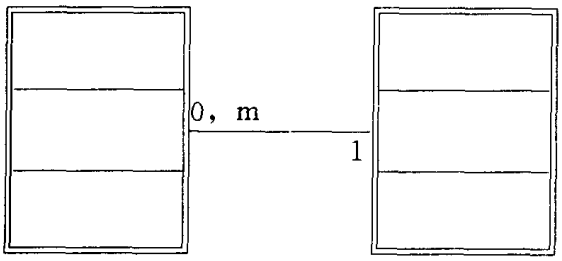


图6.11 实例连接的表示符号

在标识实例连接时,可以采用以下策略:

① 检查以前相同和相似问题域的面向对象分析结果,哪个实例连接可以直接重用? 可以吸取什么样的经验和教训?

② 为每个对象增加实例连接到其它对象,以反映问题域和系统责任内对象间的映射,到一般/特殊结构的实例连接应尽可能地连接到最适合的高层。

③为每个对象定义从它出发的连接约束的量或范围,如果是一个固定的值,则用一个数字表示;否则需确定一个范围的上下限。值得注意的是,如果连接对于该对象是任选的,则下限为

0。

注意到我们前面以类似的方式约束了整体/部分结构，在整体/部分结构和实例连接之间的差别在于它们的语义强度不同：前者是人类思维的基本方法之一，而后者仅仅是问题域中对象间的映射，因而从某种意义上来讲，前者的语义强度要比后者强得多。

4. 检查特例

当增加属性和实例连接时，需要考虑以下有些特殊情况：

(1) 属性特例

① 带有“非法”值的属性

正如本节前面提到的那样，如果某个属性有时有合适的值，而有时却没有，则应该用一般/特殊结构的策略再次查看是否还存在一个一般/特殊结构。例如，在“车辆”类中，属性“动力”可能有如下的值，如内燃机、蒸汽机或电机，但也存在该属性完全不适合的场合，如一台拖车根本就没有动力可言。这时就应该用一般/特殊结构策略检查尚不在该模型中的一般/特殊结构。

② 单属性的类及对象

人类思维的基本方法之一就是“对象及其属性”，然而当遇到只有一个属性的类及对象时，可能是下列情况之一：

· 它是问题域中某事物的一个抽象，相应的系统责任只包括一个属性，这种情况没有问题；

· 它是另一个类及对象的一个属性，不适合于该模型。

例如，类及对象“职员”只有唯一的属性“合法的名字”，如果这足以反映系统对“职员”的要求，则一切正常；但是当一个属性不适合于该模型时，则需将该属性放到它所真正描述的类及对象中，以简化模型本身。

③检查重复的属性值

检查每个重复的属性值，如果某个属性有重复值，利用“标识类及对象”策略可以发现更多的类及对象，这些类及对象的属性除了有重复值的属性外，可能还会有其它附加的属性。

(2) 实例连接的特例

① 检查每个多对多的实例连接

多对多的实例连接在每个连接端都有大于 1 的上限，对于这样的连接，检查什么属性能够描述它。例如，考虑在车辆和所有者之间的连接，属性“日期”和“数量”描述了某一时刻车辆和所有者之间的交互。针对这种情况，应该在该模型中增加一个“购买事件”类及对象。要注意的是：最终结果并不是排斥所有的多对多连接，相反是从模型中根据该模式确定更多的类及对象。

②检查同类对象之间的实例连接

某个实例连接可能发生在同一个类的对象之间，对于这样的连接，检查什么可以用来描述其映射关系。如果某个映射关系比较简单，则只要用该实例连接加上它在类及对象规范的“附加约束”一节中的描述就足够了；否则应该检查一下是否需要一个新的类及对象，以便得到该映射的更多细节。例如，“人”和“人”之间可以通过“婚姻”关系结成夫妻，如果系统需要记住“结婚”这个事件，则需用一些属性(如日期、地点、结婚证号等等)来描述。

③ 检查对象间的多重实例连接

对象间的多重映射意味着语义上的差别，这些特征可用另一个“需记忆的事件”类及对象

来描述。在图 6.12 的实例连接中，假定某个“LegalEvent”对象知道它到每个“Clerk”对象的连接，这些连接相应于“读”和“修改”。对于某个不仅负责映射，而且也区分“读”和“修改”操作的系统，还需要一个“AccessEvent”类及对象，以便能描述在“LegalEvent”和“Clerk”之间不同映射的附加语义。

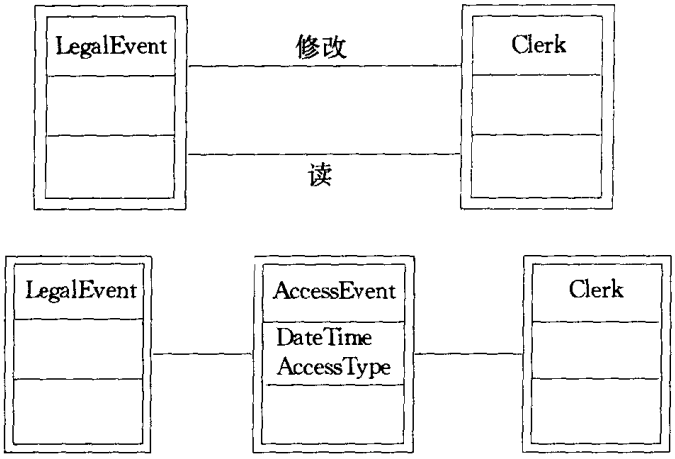


图6.12　对象间的多重实例连接

④ 检查所需的附加实例连接

考虑每一对对象，并询问以下问题：

· 该映射是否在问题域中？是否在系统责任内？

· 是否该映射在没有其它实例连接时仍然是合适的？如果是合适的，则要增加所需要的连接；否则，不需要增加另一个实例连接，以保持模型的简单性。

5. 指定属性

仔细地为每个属性命名，使用问题域和系统责任的标准词汇，使用可读性好的名字，然后给每个属性加上一到两行简短的描述。

在权衡利弊后，考虑为每个属性说明增加约束。如果某个约束能简化或减少服务说明的工作量，则值得增加。同分析有关的附加约束有：

· 度量单位、范围、限制和枚举值；

· 精度；

· 缺省值；

· 需要为该属性赋一个缺省的值吗？

· 在什么条件下允许“创建”和“访问”服务？

· 其它属性的取值对该属性有何影响？

· 该属性如何追溯到项目的需求文档上？

· 该属性可用于对象的哪些状态之中？

以下是指定属性的一个例子：

规范　传感器

属性　型号：制造厂家与型号

初始序列：初始化序列

变换：由比例因子、误差和测量单位组成

间隔:采样间隔

地址:传感器安放的位置

阈值:报警阈值

状态:操作状态(关闭、等待、监控)

值:最近读到和转换的值

通过定义属性,进一步明确了类及对象的含义,丰富了类及对象的内容,并使对象具有了为系统保存信息的能力。

6.6 定义服务

面向对象分析的一个主要目标就是分析员必须提供系统处理和流程需求的详细描述。尽管我们早已认识到这些极不稳定的目标的重要性,但还是推迟了讨论。分析员应首先集中于类及对象、结构和属性,然后再着手考虑服务,而不是直接进入到功能和流程的研究。

在面向对象分析中,服务是指某个对象所具有的特定的行为,定义服务的中心问题是定义所要求的行为。一般有三种最常用的行为分类方法:

① 基于直接的因果关系;

② 基于相似的进化历史;

③ 基于相似的功能。

这些原则被应用于本节所提出的策略之中,特别要指出的是:

①“对象状态”建立在“进化”原则的基础之上;

②“所要求的服务”建立在“相似的功能”和“因果关系”之上。

定义服务时的第二个问题是定义对象之间的必要通信,就像人同系统交互时通过命令和请求一样,在面向对象分析模型的各部分之间也使用完全相同的交互模式。

通过在类及对象规范中说明服务和消息连接,可以建立可观察、可度量的处理需求。

6.6.1 为什么要定义服务

服务进一步细化了现实世界的抽象表示,它表明某个类的对象能提供何种行为。一般地,每个数据处理系统必须有“数据”和“处理”两部分,前一节我们已经讨论了“数据”,在本节我们将主要讨论在这些“数据”之上的“功能处理”。

6.6.2 如何定义服务

定义服务包括下面几个活动:

① 标识对象状态;

② 标识所要求的服务;

③ 标识消息连接;

④ 指定服务;

⑤ 汇集面向对象分析文档。

某些分析员比较愿意先从属性入手,然后再考虑服务,而另一些则宁愿从服务到属性。我们认为这两种方法均有价值,因此在面向对象分析中我们倾向于采用两种方法混合的方式来

工作，但为了叙述的方便起见，本章还是按从属性到服务的方式来组织的。

服务放在类及对象或类符号的底部，如图 6.13 所示。

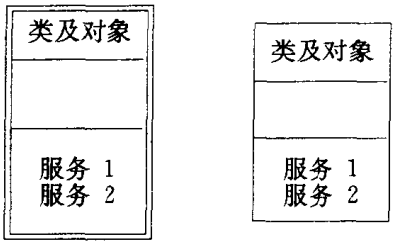


图6.13　服务的表示符号

1. 标识对象状态

一个对象从创建之始到释放(删除)为止，要经历不同的状态。对象的状态是由其属性值表示的，属性值的每次修改反映了状态的变化。

对象状态是属性值的集合，反映了对象行为的改变。为了识别对象的状态，需要检查属性的潜在取值，然后再确定这些潜在取值对应的不同行为是否属于系统责任之内。同时检查以前相同和相似问题域的面向对象分析结果，哪些对象状态可以直接重用?在标识对象状态方面可以吸取什么教训?

例如，考虑传感器系统中的属性，其中哪个属性值能反映对象行为的变化呢? 这些属性是型号、初始化序列、转换、取样间隔、地址、阈值、状态和当前值，究竟哪个值隐含了行为的变化呢? 这完全依赖于系统责任。在本系统中，只有名为“状态”的属性能反映行为的变化，其值为“关”、“等待”和“开”。

对象状态图表示一个对象在不同时期的不同状态或模式，该图标识出对象的不同状态以及状态之间的转换，如图 6.14 所示。详细的行为和行为变化将在服务规范中定义。

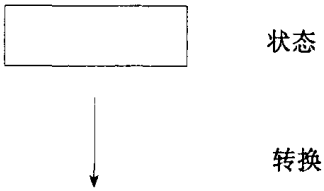


图6.14　对象状态图符号

2. 标识所要求的服务

下面我们说明如何标识所要求的服务——算法简单的服务和算法复杂的服务。

(1) 算法简单的服务

算法简单的服务适用于模型中的每一个类及对象，并且它们遵循同一个基本模式。例如，创建(Create)一个传感器对象：

- 检查型号的值；
- 检查初始化序列的值；
- 检查转换的值；
- 检查取样间隔的值；
- 检查地址的值；

- 检查阈值；
- 如果以上一切均正常，则建立和初始化一个新对象；
- 返回结果。

这种服务可以指定并处理为隐含服务。这意味着它们并不会显式地出现在面向对象分析的服务层上，而且描述隐含服务之间交互的消息连接也被隐含在服务层。

四种典型的算法简单的服务是创建(Create)、连接(Connect)、访问(Access)和释放(Release)。

Create——该服务建立并初始化该类的一个对象。

Connect——该服务将一个对象与另一个对象连接起来，实际上，该服务建立了两个对象之间的映射。

Access——该服务取出或设置一个对象的属性值。

Release——该服务释放(解除连接并删除)一个对象。

(2) 算法复杂的服务

算法复杂的服务主要分为两类：计算(Calculate)和监控(Monitor)。

Calculate——该服务根据一个对象的属性值计算一个结果。

Monitor——该服务监控一个外部系统或设备。它处理外部系统的输入和输出，负责设备的数据获取和控制。也许还需要一些其它服务的配合，如初始化(Initialize)或中止(Terminate)。

在确定一个对象除隐含服务以外还需要什么样的服务时，请从这两类服务中寻找。考察对象的状态并考虑下述问题：对象负责在其值进行何种计算？对象为了探查和响应外部系统或设备的变化需要进行何种监控？

除此之外，检查以前相同和相似问题域的面向对象分析结果，哪些算法复杂的服务可以直接重用？在定义算法复杂的服务时可以吸取什么教训？

在服务命名时，尽量使用同领域有关的名字，如对于一个传感器的监测，服务名用MonitorForAlarmCondition，而不要简单地用Monitor；如财务系统中的计算费用，服务名用CalculateFee，而不要简单地用Calculate。

3. 标识消息连接

消息连接表达了对象之间的处理相关性，是指一个服务为了完成其处理功能，而向另一个对象发出的消息请求，形象地把这两个对象称为消息的“发送者”和“接收者”，所需的处理在发送者的服务规范中命名，并在接收者的服务规范中具体定义。消息连接是在属性的强制封装性和处理这些属性的服务之间建立必要而有限的接口。

消息连接完全是为了服务而存在的。在指定服务之前，考察消息连接有助于分析员确定该对象同系统其它部分的处理相关性，然后再去考虑该对象本身的行为。头一遍扫描消息连接可以抓住许多关键的处理相关性，在实际指定服务的过程中，可望得到某些附加的消息连接。

消息连接的符号表示如图6.15所示，箭头从发送者指向接收者，箭头表示发送者“发送”了一条消息，而接收者“收到”了该消息，并且采取了某些行动以后将结果返回给发送者。箭头的每一端通常都连接到一个对象(偶尔也会连接到一个类，如创建某个类的一个新对象)以表示实际的参与者。

在标识所需要的消息连接时，针对每个对象考虑下列问题：

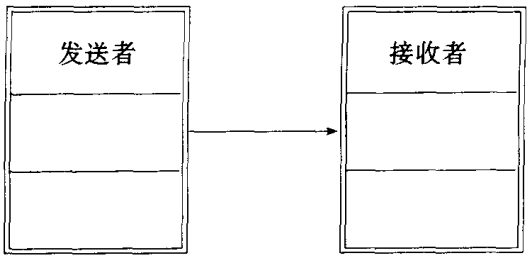


图6.15　消息连接的表示符号

① 它需要哪些其它对象提供的服务？画一个箭头到这些对象中的每一个；

② 哪些其它对象需要它提供的服务？从这些对象中的每一个画一个箭头到所考虑的对象；

③ 沿着消息连接到下一个对象，并重复同样的问题；

④ 检查以前相同和相似问题域的面向对象分析结果，哪些消息连接可以直接重用？在定义消息连接时可以吸取什么教训？

4. 指定服务

最后，对这个服务给出规格说明，并用服务流程图或其它类似的说明手段来表示服务的处理过程。

6.7　面向对象分析文档

6.7.1　文档内容

一份比较完整的面向对象分析文档包括以下内容：

①五层面向对象分析模型：主体、类及对象、结构、属性、服务；

②类及对象规范说明；

按照与下面类似的模板形式对每个类及对象进行说明：

Specification

attribute

attribute

attribute

externalInput

externalOutput

ObjectStateDiagram

additionalConstraints

notes

service 〈name & servicechart〉

service 〈name & servicechart〉

service 〈name & servicechart〉

End Specification

③必要的补充文档。

6.7.2 模型检查

使用模型检查以便及早发现错误、不一致性及不必要的复杂性。这些原则可以分为两类，一类是警告(允许被破坏的原则)；另一类是错误(不允许被破坏的原则)。

① 每个类及对象应该具有：

- 一个名字；
- 整个模型范围内唯一的名字；
- 一个以上的属性；
- 至少一个实例连接
- 至少一个消息连接
- 唯一内部属性名；
- 唯一内部服务名。

② 每个模板应该具有：

- 每个属性的规范说明；
- 每个服务的规范说明；
- 同各层相一致的内容；
- 与规范说明中每个命名的输入/输出相对应的一个用法。

③ 每个一般/特殊结构应该具有：

- 每层包含一个以上的属性或服务；
- 2—4层(否则就过于复杂)；
- 父类和子类中的属性名唯一；
- 不跨整个特殊层的属性和服务名；
- 网络结构的每一部分的一般类中包含唯一的属性和服务名。

第七章　面向对象设计

7.1　从OOA到OOD

OOA是针对问题域和系统责任的，不考虑与实现有关的因素。OOA模型由五个层次组成，即类及对象层、结构层、主题层和服务层，对应着分析工作的五个活动。OOA的各层模型化了“问题空间”。

从OOA到OOD是一个渐进的模型扩充过程。OOD针对与实现有关的因素继续运用OOA的五个活动，它包括问题域部分、人机交互部分、任务管理部分和数据管理部分等四个部分的设计。如图7.1所示，OOD模型从横向看是上述四个部分，从纵向看每个部分仍然是五个层次。

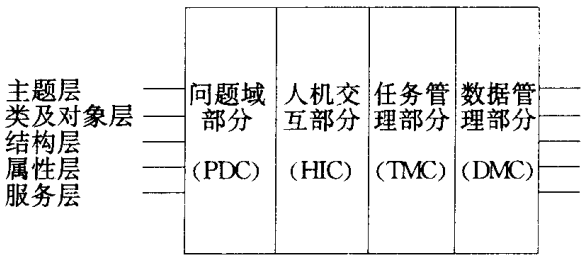


图7.1　OOD模型

从OOA到OOD不存在转换问题，而是同一种表示方法在不同范围的运用。OOD也不是对OOA模型进行细化。它的问题域部分是把OOA模型直接拿来，针对实现的要求进行必要的增补和调整，例如，需要对类、结构、属性及服务进行分解和重组。这种分解是根据一定的过程标准来做的，标准包括可重用的设计与编码类，把问题域专用类组合在一起，通过增加一般类来创立约定，提供一个继承性的支撑层次改善界面，提供存储管理，以及增加低层细节等。

OOD的其它三个部分则是OOA阶段未曾考虑的，全部在OOD阶段建立。

人机交互部分包括有效的人机交互所需的显示和输入，这些类在很大程度上依赖于所用的图形用户界面环境，例如X Window，Windows，Presentation Manager，MacApp或Smalltalk，而且可能包括“窗口”、“菜单”、“滚动条”、“按钮”等针对项目和针对GUI的特殊类。

任务管理部分包括任务的定义、通信和协调，以及硬件分配、外部系统及设备约定，可能包括的类有“任务”类和“任务协调”类。

数据管理部分包括永久数据的存取。它隔离了物理的数据管理方法，无论是普通文件、带标记语言的普通文件、关系型数据库、面向对象数据库等等。可能包括的类有“存储服务”类，协调每个需永久保存的对象的存储。

对OOA和OOD都用一种统一的基本表示，在设计开始之前，分析不必圆满地完成，这样是符合人类认识世界和改造世界的基本过程的。因为一旦你认识到事物是复杂多变的，而你又不能无所不知时，你就会看到“圆满”完成任何事的可能性都很微小。在具体开发项目时，项目

组可以运用不同的开发过程模型：

① 瀑布式

· 分析；

· 设计；

· 编程。

② 螺旋式

· 分析，原型开发，风险管理；

· 设计，原型开发，风险管理；

· 编程，原型开发，风险管理。

③ 渐进式

· 少量的分析；

· 少量的设计；

· 重复。

在OOA和OOD的表示法和策略中没有任何妨碍上述方法的地方。OOA和OOD既可以顺序进行，也可以交替进行（一个，另一个，多次重复），因此，无论是瀑布式、螺旋式还是渐进式的开发模型它都能适应。

7.2 问题域部分的设计

在OOD中，OOA的结果恰好符合OOD的问题域部分，OOA的结果就是OOD多部分模型中的一个完整部分，而且，分析结果可以被改动和增补，但这并不意味着在分析和设计之间会出现那条宽大的、不可回溯、不可追踪的鸿沟。

首先，对OOA的结果的改动和增补并不意味着把分析结果束之高阁，像变戏法似地突然进入设计；也不是建议粗粗地看一遍分析结果，做一番对分析小组的精妙或不精妙的评论，然后就走开，去做设计和编程的实际工作；而是就在OOA的基础上开始设计。

其次，改动和增补OOA的结果是把保持设计和编程与问题域尽可能相近的愿望转换成一种修改准则，即为解决一个特定的设计考虑所需要的实际变化。这可能需要分解或重组一些类及对象、结构、属性以及服务，这种扩充是真正的设计问题，而且仅当基于特定的客观标准时才是合情合理的。

7.2.1 为什么需要问题域部分的设计

面向对象的范型，促使人们按照问题本身组织系统框架。这种方法中从分析到设计再到编程的踪迹是很清晰的，因为每一阶段都是按照问题域本身的样子组织的。

因为需求总是处于不断的变化中，要使系统能从容地适应变化的需求，保持总体结构的稳定性就显得格外重要。我们需要的是分析、设计、编程结构的长期稳定性。诚然，细节会发生变化，这里增加一个类说明，那里增加一个属性或服务，但基于问题域的总体结构框架是可以长时间保持稳定的。

这种稳定性是一个问题域中的系统或者相似问题域中的系统之间的分析、设计及编程结果可以重用的关键。这种稳定性也是为更好地支持一个成功的系统超出其生命期的可扩充性

(即增加和减少其它功能)所需要的。

同时,OOA 和 OOD 结果的稳定性为评估所建议的修改带来的影响提供了清晰的基础。因此,在问题域部分中进行任何 OOA 结果的修改必须经过仔细地检查和验证。

7.2.2 如何进行问题域部分的设计

把 OOA 的结果直接放到问题域部分中,可能要进行某些修改,这就是对在 OOA 期间需求说明的修改。有些改动是由于需求本身发生了变化——客户的变化、市场的变化以及对系统误解所作的修正,其它修改是由于分析员或问题域专家在理解上的欠缺。作为一个实习的专业人员,出现这种情况是很正常的。

在上述各种情况下只需改变 OOA 的结果(也许你的单位随之进行的是通常的变更管理和需求控制),然后使它们自动地反映到 PDC 中。

除此之外,问题域部分的设计要针对特定的实现环境,对 OOA 的结果加以增补,要考虑以下因素:

·重用设计和编程类;
·把问题域专用类组合在一起;
·通过增添一般类而建立协议;
·调整继承的支持级别;
·改进性能;
·提供数据管理部分;
·增加低层细节;
·不要仅为反映队伍分工而作修改;
·复审并挑剔对 OOA 结果的增补。

1. 重用设计和编程类

首先考虑如何从自己的或别人的源程序中把现成的类增加到问题域部分。现成的类可能是用 OOPL 写的,也可能是用某种非 OOPL 写的可用软件,在后一种情况下,把软件封装在一个特意设计的、基于服务的界面中,改造成类的形式,把现成的类增加到问题域部分中。

其次,划掉现成类中任何不用的属性和服务,并增加一个在现成类到问题域类之间的一般/特殊关系。

接着,划掉问题域类中不再需要的部分,这些属性和服务现在是从现成类中继承的了;并修正问题域类的结构和连接,必要时把它们移向现成类。

2. 把问题域专用类组合在一起

在 OOA 中,没有引入一个类放在所有类的上方。在 OOD 中,通常先引入一个类以便把问题域专用的类组合在一起,它仅仅起到"根"类的作用,把全部下层的类组合在一起。为什么需要这样做呢?因为它可以形成一个一般/特殊结构,从而概括 OOA 模型中几乎每一个类及对象。而且,权衡模型的可理解性和模型的语义内容,也缺少这样一个附加的"根"类。

如图7.2所示的例子中的"OOARoot"就是这样的一个"根"类。

当没有一种更满意的组合机制可用时,这实际上就是一种把类库中的某些类组织在一起的方法。而且这样的类可以用于建立一个协议(接下来讨论)。

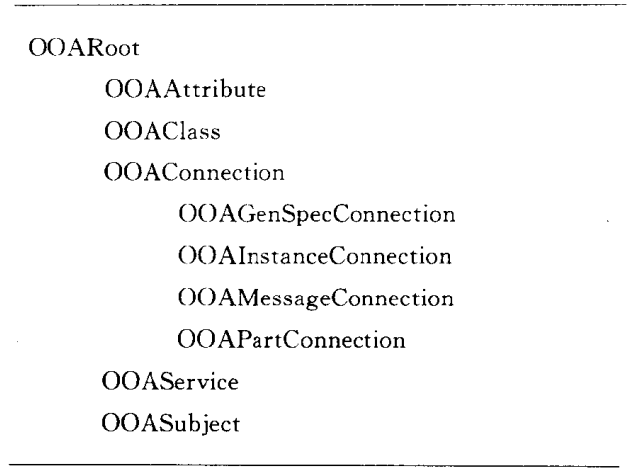


图7.2 把问题域专用类组合在一起

3. 通过增添一般类而建立协议

有时，一些专用类将需要一个相似的协议，这意味着它们将要定义一个相似的服务（以及相应的属性）集合。在这种情况下，可引进一个附加类，以便建立这种协议——服务的公共集合命名——其细节在特殊的类中定义。在上面例子中的“OOAPart”也是这样的一个类，它为所有的特殊类定义了一个共同遵守的协议。

4. 调整继承的支持级别

如果OOA的一般/特殊结构包括多继承，在使用一种只有单继承或无继承性的编程语言时，就需要对OOA的结果作一些修改。

（1）多继承模式

考虑图7.3所示的多继承模式，这个多继承模式可称为狭义的菱形。这个模式中属性和服务命名冲突比较频繁，使用者必须注意到这种冲突。

第二种多继承模式可称为广义菱形。这里，菱形开始于比Person和Role类更高的某个地方——在最高的一般类，即通常称为“根”类的地方。这里，属性和服务命名的冲突比较少，但它需要更多的类来表示设计。如图7.4所示。

（2）针对单继承语言的调整

对单继承语言，可用两种方法从多继承结构转换为单继承结构。

① 分解多继承，使用它们之间的映射。这种方法把多继承模式分为两个层次结构，使用它们之间的映射，即用一个整体/部分结构或者一个实例连接。

实际上，这种方法是把若干特殊类的对象模拟成一般类的若干对象扮演的一些“角色”，分别如图7.5和图7.6所示。

② 展平为单继承。这种方法把多继承的层次结构展平而成为一个单继承的层次结构，如图7.7所示。这意味着，有一个或者多个一般/特殊结构在设计中就不再那么清晰了。同时也意味着，有些属性和服务在特殊类中重复出现，造成冗余。

（3）针对无继承性语言的调整

程序设计语言中继承的意义远不只是在语法上带来一些好处，它提供了表达问题域一般性——特殊性语义的语法；它明确地表示了公共属性和服务；它还为通过可扩充性而达到可重

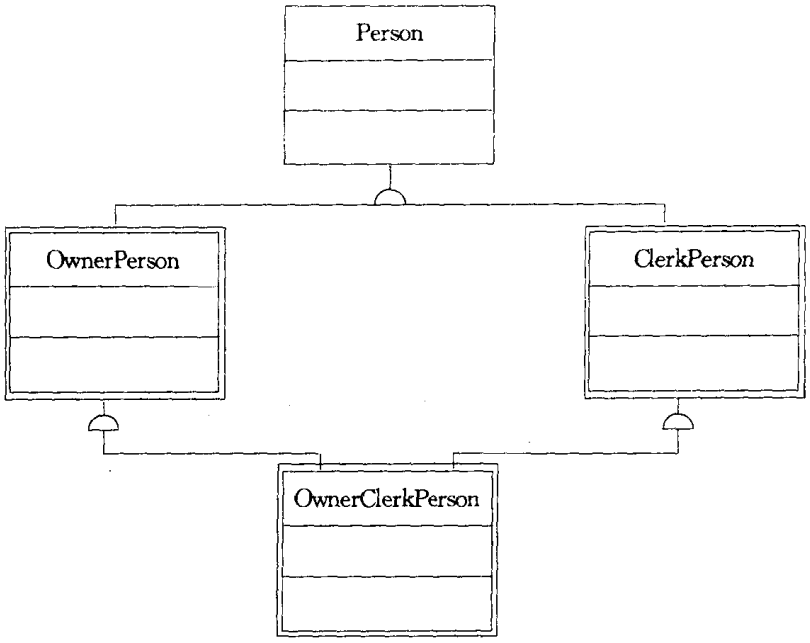


图7.3 一个多继承模式——狭义的菱形

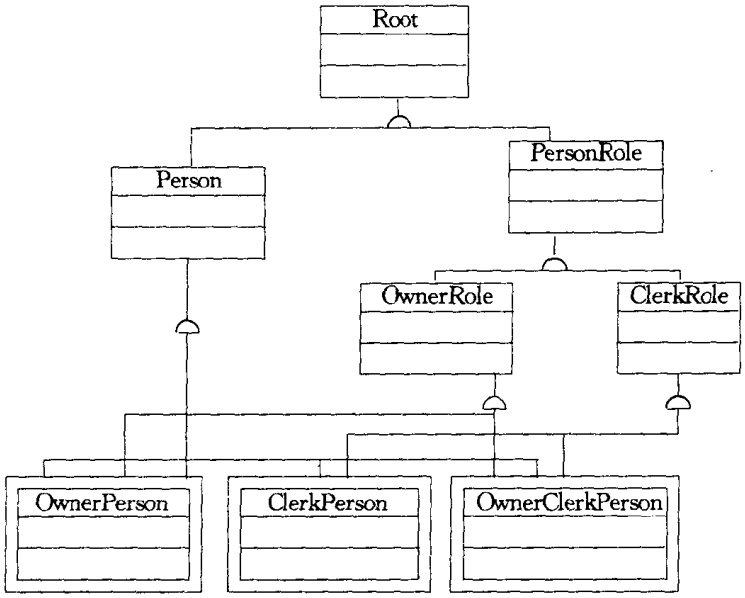


图7.4 另一个多继承模式——广义的菱形

用性提供了基础。

但是，由于在开发组织方面的各种理由，有些项目选择的路线可能是从OOA到OOD，然后到一个无继承性语言。

对一个无继承性语言来说，需要把每个一般/特殊结构的层次展开，成为一组类及对象——尽管这也可以由编程语言自己来做，那就是运用命名惯例把它们组合在一起，而不是明确地在OOD模型的这一时刻来做此事。

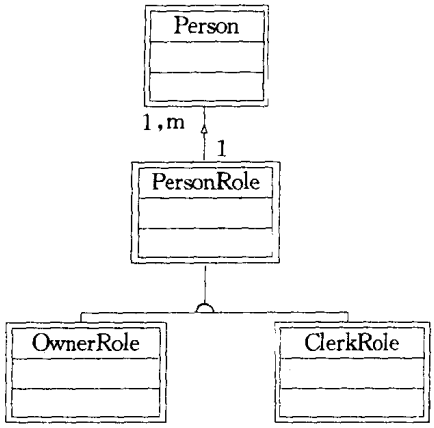


图7.5 针对单继承的调整——分解多继承，采用整体/部分结构映射

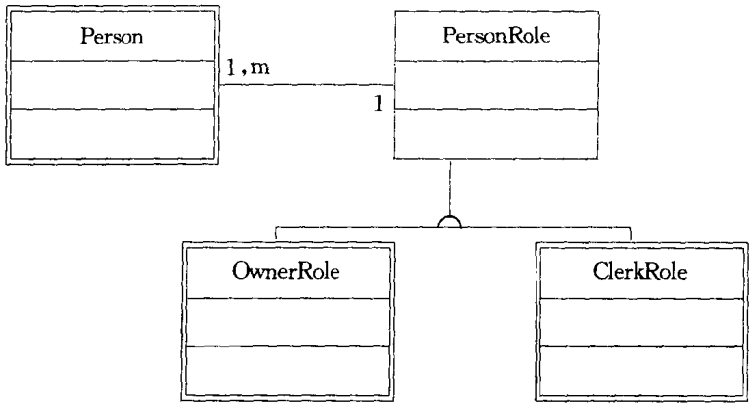


图7.6 针对单继承的调整——分解多继承，采用实例连接映射

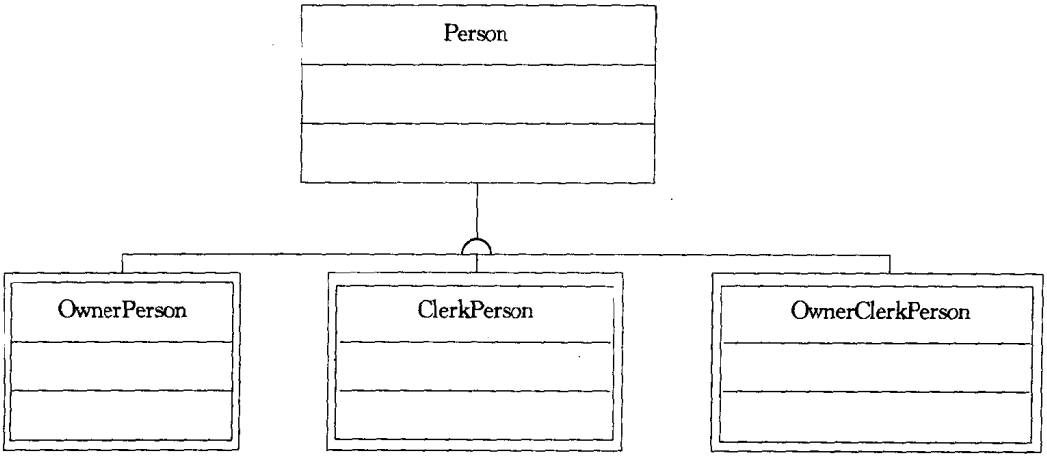


图7.7 针对单继承的调整——展平为单继承

5. 改进性能

尽管现在用的机器越来越快，但不管将来的机器多么快，软件开发人员总会用尽机器的最后一个字节的内存和最后一个微秒的时间，因此性能仍然是一个系统成功的关键因素。性能的含义远不只是一个系统或应用程序执行得多快，上乘的软件能把该做的事情做得“足够快”（符合需求或客户期望），而且费用和进度又符合要求。

下面有几种可望改进性能的措施：

① 在对象之间具有高度繁忙的消息流通的情况下，这种高度耦合可能暗示需要把两个或更多的类加以合并；

②避开正常的数据抽象原则而允许服务偷偷地从其它对象中强行获得属性值；

③在类及对象中扩充一些保存临时结果的属性。

6. 提供数据管理部分

为了提供数据管理部分，每个被保存的对象需要知道自己是怎样被存储的。

(1)第一种方法是“每个对象自己保存自己”：

①通知一个对象保存自己；

②每个对象知道如何保存自己；

③增加一个属性和一个服务来完成此事。

(2)第二种方法是，每个对象把自己传送给数据管理部分，让数据管理部分来存储对象自己：

① 通知一个对象保存自己；

② 每个对象知道为了保存自己的状态应该传送什么消息到数据管理部分，例如“store me”。这叫做“基于媒介”的存储方法。在这种方法中，可以把任何一个提供同一组服务的存储系统插入数据管理部分，而不需对问题域部分做额外的修改；

③ 增加属性和服务以完成此事。

无论第一种还是第二种方法，所需的属性和服务如下：

① 对于单继承环境，直接修改问题域部分

· 增加一个属性来标识对象属于某个特定类，例如增加一个 ClassName 属性；

· 增加一个服务来定义对象如何存储自己的值；

· 隐式地保持它们，不是包含在图中，而是在每个类及对象的说明部分定义它们。

② 对于多继承环境

· 把增加的东西放在一个新类中，然后修改每个带有可存储对象的类，使之继承这个附加的一般类；

· 增加一个属性来标识对象属于某个特定类，例如增加一个 ClassName 属性；

· 增加一个服务来定义对象如何存储自己的值；

· 隐式地保持继承，不是包含在图中，而是在每个类及对象的说明中定义。

(3)第三种方法是，每个需要长期保存的对象由一个面向对象数据库管理系统(OO-DBMS)来管理：

①每个对象都由 OO-DBMS 来管理，而这个 OO-DBMS 负责为可考虑中的系统存储和检索对象；

② OO-DBMS 为维持贯穿设计和编程的问题域结构框架提供了最好的潜在可能性。但是

采用其它的方法也是明智的。

数据管理部分本身将在后面的章节中详细介绍。

7. 增加低层细节

为了设计和编程的方便，可以在低层成分中分离出一些独立的类，要这种方法有助于把与机器细节有关的东西放在低层类中隔离起来。

8. 不要仅为反映队伍分工而作修改

在进行小组分工时，不要分裂问题域的基本结构和类，要遵守力求稳定性、可重用性和可扩充性的原则，那种因人设事的修改是毫无意义的。相反，应完整地保持问题域部分，把较大的结构和类分配到各个小组。

9. 复审并挑剔对 OOA 结果的增补

考察所做的选择，重新审查和挑剔对问题域部分的内容所作的任何 OOD 修改。无论何时何地，要尽可能地保持 OOA 结果中建立的基本问题域的结构。

7.3 人机交互部分的设计

OOA 部分的主要任务是了解问题域和用户需求，现在需要进一步考虑人机交互部分，包括窗口内容和报表的设计格式。原型有助于帮助实际交互机制的开发和选择。

采用多层次、多部分模型有助于把与实现有关的人机交互部分从面向对象分析及需求工作中分离出来。这种分离提高了可移植性，并减少了由于实现人机交互本身的构造技术发生变化而引起的变化影响。

7.3.1 为什么需要人机交互部分

人机交互部分突出人如何命令系统以及系统如何向用户提交信息，人在使用计算机过程中的感受直接影响到他(她)对系统的接受程度。随着计算机应用的不断普及和深入，非计算机专业人员在使用计算机的人群中所占的比例也在不断增加，人机交互部分的友好性直接关系到一个软件系统的成败，虽然好的人机交互部分不可能挽救一个功能很差的软件，但同时性能很差的人机交互部分将使一个功能很强的产品变得不可接受。

人机交互中起主导作用的是人，为了考察其产品的用户友好性，国外一些大的软件公司在新产品上市之前，总要组织一些实际的用户进行产品试用，并通过详细考察和记录被试验者的生理反应，从而确定产品是否使用户感到满意。

7.3.2 如何设计人机交互部分

设计这一部分的策略由以下几点构成：

- 对人分类；
- 描述人和他们的任务脚本；
- 设计命令层次；
- 设计详细的交互；
- 继续做原型；
- 设计人机交互部分的类；

· 根据图形用户界面(如果可用的话)进行设计。

这些活动的次序反映了在人机交互设计中正常的优先事项集合:首先是人,其次是任务,然后是工具。

1. 对人分类

花些时间去研究将使用系统的人,把自己置身于别人的地位,并在那里多待一会儿。身临其境地看人们如何实际地做他们的工作。这是绝对需要的,不会有合同方面或社会方面的障碍阻拦这种学习。而且,这种努力需要真正设身处地为那些与系统利害相关的人着想。

考虑这些人想达到什么目的?他们要完成什么任务?你能提供什么具体工具来支持那些任务?工具如何做得最协调?

一开始,把人分为不同的类型。如果在问题域部分中有一个一般/特殊结构"Person",就用它的一般/特殊模式作为开始点。例如,在汽车注册发照系统中:

Person/Clerk/Owner/OwnerClerk

然后考虑增加与系统交互的人的子集。考虑按以下的一个或几个原则分类:

(1) 按技能层次分类

初学者/临时人员/中级水平/高级水平

(2) 按组织层次分类

行政人员/办公人员/职员/管理人员/办事员

(3) 按不同组的成员身份

职员/顾客

2. 描述人及其任务脚本

对以上定义的每一类人,考虑下述问题并制表:

谁

目的

特征(年龄、教育水平、限制等)

关键的成功因素

必须/想要

喜欢/不喜欢/有偏见

熟练程度

任务脚本

对每一类人,把这些问题搞清楚,并写下其中每一个人的情况。这里有运用这种方法的一个例子:

谁:我是一个分析员

目的:我想做实际的分析工作。给我一个工具,它能帮助我更有效地工作(并卸下我画图和检查这些图的重担,否则简直要累垮了)

特点:

年龄:42岁

教育水平:大学毕业

限制:我不喜欢微型打印,任何小于9个点的打印都太小

关键的成功因素:

我想做分析工作。工具必须使我做有效的分析畅通无阻

给我一条捷径，使工具不与我正在做的工作冲突。把它做得有趣一些，我希望工具能捕捉假设和思想，并能实时地进行折衷

我希望能在任何时刻给出模型任何部分的文档。我认为这种信息和需求本身一样重要

熟练程度：高级熟练程度

任务脚本：

识别“核心”的类及对象

然后识别“核心”结构

自始至终，发现了属性和服务就把脑子里想到的东西加进去，但要到后来的工作中才想在模型中看到它们

检验模型：

打印模型及其全部文档

3. 设计命令层

现在是设计命令层的时候了，包括下列工作：

· 研究现有的人机交互活动的寓意和准则；

· 建立一个初始的命令层；

· 细化命令层。

(1) 研究现有的人机交互活动的寓意和准则

如果命令层将放在一个已建立的交互系统之中，就从研究现有的人机交互活动的寓意和准则开始入手，对于图形用户界面更是如此。

这种准则可能是非正式的(例如，“它看起来以及在感觉上就像这样”)，也可能是正式规定的(例如，Apple 计算机公司为 Macintosh 制订了相当广泛的准则)。在上述各种情况下，人机交互活动的寓意和准则都是随时间演化的，因此需要继续观察、研究、做原型，并吸取好的例子。

(2) 建立一个初始的命令层

命令层可能以多种形式呈现给用户：

· 命令行；

· 一个菜单条；

· 一系列图标，当点中它们时会激活一个动作。

细心的读者可能会注意到，命令层是用过程抽象组织可用服务的一种体现。确实，在这一点上，设计转到了如何体现用过程抽象组织所需的服务。

从服务的基本过程抽象开始(如图7.8所示的一个例子)，并修改它以符合你的特定需要：

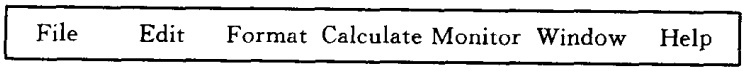


图7.8 初始的菜单条

(3) 细化命令层

为了细化命令层，要考虑排列、整体/部分组合、宽度与深度的对比、最少操作步骤等问题。

① 排列

在开发这个层次时，要细心地选择醒目的服务名，并在这个层次的各部分合理安排服务名：把使用最频繁的服务放在前边；按照用户的工作步骤排列。

② 整体/部分组合

通过服务本身发现整体/部分模式，以帮助在命令层中对服务进行组织和分块。

③ 宽度与深度的对比

工作准则是不要超过人类短期记忆的局限。Miller 博士在《魔数7±2：我们处理信息的能力局限》一文中指出，人类的短期记忆能力限制在同时能记住5－9件东西。Miller 于1975年又在《十五年后的魔数7》一文中又重新考虑了此事，他说7±2原则不如改为每次记忆3块，每块至多有3项。

应用 Miller 的方法，查看命令层中宽度与深度的对比，努力寻求每块有3项的3个块的宽度，把深度限制在大约3层左右，这样用户就不必为知道他进入应用有多远而大伤脑筋了。

④ 减少操作步骤

使点取、推动以及键盘操作减少到最少而能完成用户所要求的工作。同时，为系统的高水平用户提供操作捷径。

4. 设计详细的交互

人机交互的设计有若干准则，包括：

① 一致性

采用一致的术语、一致的步骤和一致的活动。

② 操作步骤少

使敲键或鼠标点取次数减到最少，甚至要减少做某些事所需的下拉菜单的距离。

③ 不要“哑播放”

“哑播放”是一个广播术语，指的是一段没有声音的时间。这里，不要“哑播放”意思是不要让用户感到寂寞。每当用户要等待系统完成一个动作时，要给出一些反馈信息，说明工作正在进展并取得了多少进展。所有这些意味着有意义的、及时的反馈是必不可少的。

④ 闭包

用一些小步骤引出定义良好的活动，用户应该感受到他们的活动中闭包的意义。

⑤ Undo

人难免做错事，通常在这种情况下是想法恢复原状，至少是部分的恢复。

⑥ 减少人脑的记忆负担

不应该要求人从一个窗口中记忆或写下一些信息，然后在另一个窗口中使用。任何需要人按特定次序记忆的东西以及类似的东西应该组织得容易记忆。

⑦ 学习的时间和效果

保持简短，不要期望人们去读印刷文档，更不用说让人去琢磨它的奥妙和细微区别。为更多的高级特性提供联机帮助，以便人们在需要时能找到它们。

⑧ 趣味与吸引力(外观与感受)

人们喜欢使用那些使人感到有趣的软件。枯燥无味的软件也能忍受，但只有在非用不可时才会去用。

5. 人机交互部分的类

人机交互部分的类在很大程度上依赖于所选用的图形用户界面，例如 X Window，Motif，Windows，Presentation Manager，MacApp 或 Smalltalk，不同的图形用户界面具有基本的区别。例如，字型、坐标系统以及事件等。此时，需要根据选定的图形用户界面以及前面的策略设计人机交互部分的类，包括窗口、菜单、滚动条、按钮等等。

7.4 任务管理部分的设计

任务又称为进程(进程是一连串的活动，由它的代码所定义)，若干任务并发执行时叫做多任务。下列几类系统是需要多任务的：

① 负责局部设备的数据采集及控制的系统需要多任务；

② 某些人机界面——其中的多窗口可被同时选来作输入——需要多任务；

③ 多用户系统，一个用户任务可能有多份复制品；

④ 多子系统软件结构，任务可能被用作程序片之间的协作和通信；

⑤ 负责与其它系统通信的系统需要多任务。

单处理机上的多任务，可能需要一个任务在其它任务执行期间与它们协作和通信。在操作系统支持下，这些任务以时间片轮转的方式运行，产生同时运行的感觉。多处理机硬件结构，每台处理机需要独立的任务，此时要增加支持进程间通信的任务。

任务增加了设计、编码和过程的复杂性，因此必须细心地选择并作最终调整。

7.4.1 为什么需要有任务管理部分

对某些应用来说，任务能简化总体设计和编码。

独立的任务把必须并发执行的行为分离开来。这种并发行为可以在多个独立的处理机上实现，或者在运行多任务操作系统的单处理机上模拟。

有没有别的方法？一个可考虑的合理的方法是轮流地执行顺序程序；它执行一些小而快的程序块，在每个块之后，检查在它忙着的时候发生了什么事，并作出相应的反应。

另一个远为逊色的方法是把并发行为交叉地放在一个顺序执行的程序中。这是可以做到的，但是设计和编码一下子就变得笨拙了。想想用这种方法得到的程序的复杂性：它被写成一个大程序，但它的每两行源程序都需加上检验，每个检验需要查看新来的数据或请求，并开始做与此有关的工作，同时要查看其它的(可能更优先的)输入或请求，结果是程序代码如同一团乱麻。

为什么要设立任务？为了简化必要的并发行为的设计和编码。为什么有时不设立任务？为了避免画蛇添足地增加并发行为，因为这种做法会增加设计、编码、测试和维护的复杂性。

7.4.2 怎样设计任务管理部分

任务的选择和调整，遵照下述策略：

- 识别事件驱动任务；
- 识别时钟驱动任务；
- 识别优先任务和关键任务；

· 识别协调者；

· 审查每个任务；

· 定义每个任务。

这种策略的要点是识别并设计任务，加上包含在每个任务中的服务。

1. 识别事件驱动任务

有些任务是事件驱动的，这些任务可能是负责与设备、其它处理机或其它系统通信的。

一个任务可以设计成由一个事件来触发，该事件常常针对一些数据的到达发出信号。数据可能来自输入数据行或者来自由另一个任务写入的数据缓冲区。

当系统运行时，这类任务的工作过程如下：任务睡眠（不消耗处理机时间），等待来自一个数据行或其它来源的中断；当接收到中断后，任务醒来，读取有关数据，把数据放到内存的缓冲区或其它目的地，对所有需要知道此事者发出通知，然后又回到睡眠状态。

2. 识别时钟驱动任务

这类任务按特定的时间间隔被触发去进行某些处理。某些设备需要周期性地数据采集和控制；某些人机界面、子系统、任务、处理机或其它系统可能要周期性的通信。这正是时钟驱动任务的用途。

当系统运行时，这类任务的工作过程如下：任务设置一个唤醒时间，然后去睡眠；任务睡眠（不消耗处理机时间），等待来自系统的一个时钟中断；当接收到这个中断后，任务醒来，进行必要的处理，然后又回到睡眠状态。

3. 识别优先任务和关键任务

优先任务既包括高优先级的也包括低优先级的处理需要。关键任务用来分离出对于系统的成败特别关键的那些处理，这种处理通常具有特别严格的安全性要求。

（1）高优先级

有些服务是高优先级的，这种服务可能需要划到一个独立的任务中，使该服务在一个紧迫的时间约束下完成。可能需要用附加的任务把这种服务分离出来。

（2）低优先级

在优先级谱系的另一端，有些任务是低优先级的，可以进行低优先级的处理（常常称作后台处理）。可能需要用附加的任务把这种服务分离出来。

（3）高度关键

有些服务对于系统的持续操作可能是高度关键的，而且有些服务甚至在降级的操作方式期间对于持续的操作可能也是必不可少的。可用附加的任务来分离这种关键处理，这种分离隔开了高度安全性处理所需的深入细致的设计、编程和测试。

4. 识别协调者

有三个或更多的任务时，可考虑另外增加一个任务，这个任务起到协调者的作用。

这样一个任务加在上边，现场切换时间（从一个任务转到另一个任务的时间）可能使这种设计方法遇到困难，但这个任务可为封装任务之间的协作带来好处，它的行为可以用一个状态转换矩阵描述。

用这样的任务来协调其它任务，仅此而已。不要用它去实现那些本应属于分配给被协调的任务所包含的类及对象的服务。

5. 审查每个任务

要使任务数保持到最少。

只有一件事还是有多件事同时在进行?在开发与维护期间根据不同的理解足可引起争议。设计多任务协调的主要问题之一是设计者常常迷恋于定义太多的任务。例如,一个设计者可能仅仅为了在处理他自己的小小的设计与编程问题时寻求方便而增加任务,而总体设计的可理解性和技术复杂性可能要受到损失。

因此要审查每个任务,确保它能满足一个或多个选择任务的工程标准——事件驱动、时钟驱动、优先/关键任务或协调者。

6. 定义每个服务

(1) 它是什么服务

首先要说明是什么任务:为任务命名,并简要地说明该任务。为 OOD 部分的每个服务增加一个新的约束——任务名,并为该约束分配一个值。

如果一个服务被分裂,交叉在多个任务中,则要修改服务名及其描述,使每个服务能映射到一个任务。对那些与设备、其它任务或其它系统协调和通信的服务,要用协议专用的细节来扩充服务的规格说明。

(2) 如何协调任务

定义每个任务怎样协调工作。指出它是事件驱动的还是时钟驱动的;对于事件驱动的任务,描述触发该任务的事件;对时钟驱动的任务,描述在触发之前所经过的时间间隔,同时指出它是一次性的还是重复的时间间隔。

(3) 如何通信

定义每个任务如何通信。任务从哪里取数据(例如,从输入行、信箱、信号量、缓冲区或约定点)?任务往哪里送数据(例如,输出行、信箱、信号量、缓冲区或约定点)。

以上策略可以用图7.9所示的任务管理部分模板概括出来。

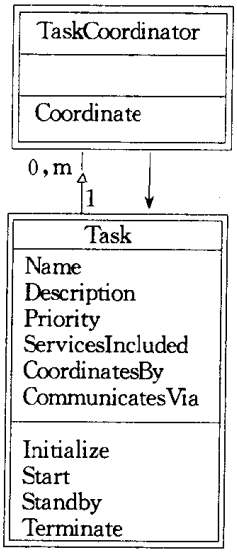


图7.9 任务管理部分的OOD表示模板

7.5 数据管理部分的设计

数据管理部分提供了在数据管理协调中存储和检索东西的基本结构。

7.5.1 为什么需要数据管理部分

数据管理部分旨在隔离数据管理方案的影响，不管该方案是普通文件、关系型数据库、面向对象数据库或其它方式的。

7.5.2 如何设计数据管理部分

设计数据管理部分既包括数据存放方法的设计，也包括相应服务的设计。

1. *数据存放设计*

按普通文件、关系或面向对象的数据管理来设计数据的存放。

(1) 采用普通文件数据管理

① 定义第一范式表：列出每个类及其属性，把这个列表引入第一范式，产生符合第一范式表的定义；

② 为每个第一范式表定义一个文件；

③ 测量性能和存储需求；

④ 然后，回到第一范式，设法满足性能及存储需求；必要时把表示一般/特殊结构的属性压缩成一个单一的文件，这样做可减少文件的数目(代价是为空域增加了一些存储空间)；必要时把一些属性组合成某些译码值，而不用分离的域来表示，这会减少所需的存储量(代价是增加了编码和解码所需的处理)。

(2) 采用关系数据库管理

① 定义第三范式表：列出每个类及其属性，把列表引入第三范式，产生第三范式表的定义(第四、五范式也很好，尽管有些深奥)；

② 为每个第三范式表定义一个数据库表；

③ 测量性能和存储需求；

④ 然后，回到第三范式，设法满足性能及存储需求。

(3) 采用面向对象的数据库管理

① 扩充的关系方法：具体做法和“采用关系数据管理”中所介绍的步骤相同。

② 扩充的OOPL方法：不再需要附加的服务。数据库管理系统本身为每个需要长期保存的对象提供了“存储我自己”的行为，只需把每个需要长期保存的对象标出来，让面向对象的数据库管理系统去负责其存储。

2. *设计相应的服务*

为每个需要存储其对象的类增加一个属性和一个服务。因为每个需要存储的对象都要使用这样的属性和服务，所以把它们作为隐含的属性和服务来处理，即，不放在OOD模型的属性层和服务层，而在相应的类及对象规格说明中描述。

采用这样的设计，一个对象知道如何存储自己。“存储我自己”的属性和服务形成了问题域部分和数据管理部分之间的必要桥梁。使用多继承时可以定义这样一个附加属性和相应服务

的类，然后由每个需要存储其对象的类来继承。

(1) 采用普通文件数据管理

对象需要知道要打开哪个(或哪些)文件，如何正确地定位到文件的有关记录，如何检索旧的值，以及如何用新的值进行更新。

定义一个名为 ObjectServer 的类及对象，它带有两个服务：

① 告诉每个对象存储自己；

② 检索被存储的对象(搜索、取值、建立初始对象)，供设计中的其它成分使用。

(2) 采用关系型数据库管理

对象需要知道存取哪些表，如何存取所需的栏，如何检索旧的值，以及如何用新的值进行更新。

定义一个名为 ObjectServer 的类及对象，它带有两个服务：

① 告诉每个对象存储自己；

② 检索被存储的对象(搜索、取值、建立初始对象)，供设计中的其它成分使用。

(3) 采用面向对象的数据库管理

① 扩充的关系方法：与采用关系型数据库管理系统时所介绍的方法相同。

② 扩充的 OOPL 方法：不需要增加属性和服务。数据库管理系统本身提供了“存储我自己”的功能，使每个对象能长期保存。只要把每个需要长期保存的对象标出来即可，至于它如何保存和恢复，则由面向对象的数据库管理系统去负责。

面向对象设计文档包括问题域部分、人机交互部分、任务管理部分和数据管理部分的设计结果。

第八章　OSA 方法简介

第六章、第七章主要讲述了面向对象的基本概念，并以 COAD-YOURDON 方法为主，介绍了面向对象分析和设计。如前所述，至今已提出了许多面向对象分析技术，但大体上可分为两大学派。一派称之谓“方法驱动的方法”（例如 COAD-YOURDON 方法），一派称之谓“模型驱动的方法”（例如本章介绍的 OSA 方法和 RUMBAUGH 等提出的 OMT——对象建模技术）。为了对面向对象方法有一个较为全面的了解，本章将简单介绍一下 OSA 方法。

OSA（Object-oriented Systems Analysis）是由 D. W. Embley 等于1992年提出的一种面向对象的系统分析方法。

OSA 不仅是面向对象的，而且还是模型驱动的。就是说，它提供了一组基本的模型化概念以及三种 OSA 模型（对象关系模型，对象行为模型，对象相互作用模型），分析员在进行系统分析中，以给定的模型化概念为指导，以模型构造（model construction）为驱动，产生对系统的询问，捕获系统的知识，建立系统的 OSA 模型。因此，对于 OSA 的了解，最重要的是了解构造系统的概念框架。

OSA 方法强调对系统的理解以及文档的组织，为此，OSA 提供了三种系统概念模型，它们从不同角度描述了所考虑的系统，形成了互补的、且统一的系统视图；对于文档的组织，OSA 还提出了一致的控制复杂性机制，最终产生的文档为以后的设计和实现提供了一致的基础。

由于 OSA 是模型驱动的，因此产生的 OSA 模型通常缺乏一定的表达能力。但是，按照 OSA 的观点，分析阶段的任务是要研究理解系统，建立其相应的文档，而不应涉及与以后设计和实现有关的许多思想。

建造 OSA 系统分析模型的过程与方法驱动的分析不同，它不是一步一步的过程，而是随着相互作用的模型化活动，并发进行的。例如，一旦标识了一个对象类，分析员可以或在当时，或在以后为该对象类建立相应的行为模型（即状态网），开发对象关系模型中与该对象类有关的部分。

尽管 OSA 并不规定 OSA 模型成分的建造次序，但是，OSA 的目标确是十分明确的，即建造能够精确反映系统的分析模型，为以后的设计和实现提供必要的理解。因此，在系统开发过程中，可以自底向上，也可以自顶向下。即是说，OSA 允许自由、灵活的系统分析风格。

8.1　OSA 的对象关系模型（ORM）

OSA 的对象关系模型（ORM）是用对象关系图表示的。ORM 主要用于记录一个系统的说明性信息，即记录有关对象及对象之间关系的信息。为此，OSA 给出了如下几个模型化概念：

·对象　　·关系　　·对象类　　·关系集合　　·约束

这些概念独立于相应的图示约定，即它们可以用其它图示符号表示。

8.1.1 基本的模型化概念

1. 对象

一个对象是一个单一的实体或概念。例如，一个人是一个对象，一个地方是一个对象，一件事情是一个对象。对象可以是物理的，也可以是概念的。一个对象可以与其它对象发生联系，也可以由其它对象组成，但是，每一对象必须是可标识的。

在自然语言中，名词或名词短语通常表示着对象。

在OSA 中，用实心点表示一个对象，并且为了标识该对象，在实心点附近给出一个或多个名词或名词短语。图8.1所示。

图8.1 对象

2. 关系

关系是对象间的一种逻辑连接(a logical connection)。例如，“John is the manager of Janet”，就是对象 John 和 Janet 之间的一个关系。

在自然语言中，动词或动词短语常常表示着关系。

在OSA 中，用一条线段表示对象间的关系，并且为了标识这一关系，给出该关系的描述，即给出关系名——通常是一个语句，以便使这一关系容易理解。如图8.2(1)所示。

为了简化关系的描述，对于常用的二元关系，可以省略关系名中所包含的对象名，只给出动词或动词短语，并在动词或动词短语下面给出一条单箭头线或双箭头线。如图8.2(2)所示。

在一个关系中，连接的数目称为该关系的“元”。在OSA 中，允许存在多元关系。在图8.2(3)中，我们给出了一个五元关系。

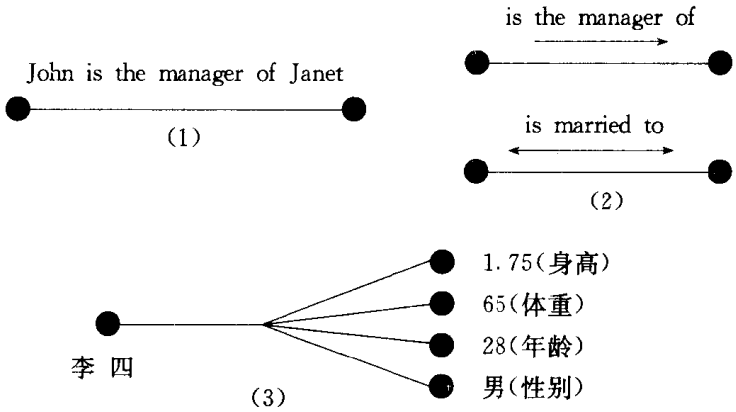


图8.2 关系

在ORM 图中，就单独的一个对象而言，看起来往往不易理解它代表什么，只有了解了它与其它对象间的关系，才能知道它所具有的语义。

3. 对象类

通常，我们按着某一逻辑理由，把一些对象分为若干个集合，以便易于处理，这一活动称为分类。对于当今所面临的、复杂的软件系统，也必须引入一种组织技术，以便控制复杂性。随之，

产生了“对象类”这一概念。

按照某一逻辑理由，把一些对象分为若干个对象集合，我们称其中任一对象集合为一个对象类。

在OSA中，用矩形表示一个对象类，其中包含该对象类的名字。如图8.3所示。命名一个对象类，通常采用一般性的、并且可以描述该对象类中任一成员的名字，参见图8.3。

OSA鼓励分析员把一些对象组织为若干个对象类，即标识对象类。对于分类规则，OSA不作任何限制。分析员可以按照自己认为有意义的逻辑理由，对对象进行分类。其中，应当考虑对象类中的对象，除了他们同属一个对象类外，一般还具有一些共同的特性；另外，还应注意使用的分类规则，应尽量避免产生通信上的更多困难。

图8.3 对象类

4. 关系集合

一个关系集合是一组关系，其中每一关系均具有相同的结构和相同的语义。即在这组关系中：

·每一关系所涉及的对象数目相同；

·每一关系所涉及的对象分别属于同一对象类；

·每一关系中对象名在关系中的位置相同；

·把每一关系名中的对象名去掉，所余部分不仅位置相同，而且是等同的。

我们可以把这样一组关系共同对应的结构看作是一块模板(template)。如图8.4所示。

在每一模板中，对象类指定了参与关系的那些对象的“源”，短语(例如：“owns”)表达了对象间的逻辑联系。

一个关系集合的命名规则是使用相应模板中出现的文字。例如，“person owns vehicle”。实际工作中，我们在说明一个关系集合时，往往首先说明了一块模板，因此，能够容易遵守关系集合的命名规则。一个关系集合可以有多个名字。

在OSA中，用菱形以及连接到相关对象类的线段来表示一个关系集合，把关系集合的名字写在菱形附近。见图8.5(1)。

对于二元关系集合的名字，由于大多数具有统一的模式，即两个对象类的名字分别位于那个短语的前后，因此，可以把二元关系集合的名字简写，如图8.5(2)，即省略了菱形，省略了对象类名，增加了一条有向箭头线。

5. 约束

通过以上给出的模型化概念，我们可以建立一个系统的对象关系模型。但是，为了使描述的系统更令人满意，常常需要对其中的对象类和关系集合增加一些约束，指出它们所具有的其它性质。为此，OSA引入了不同类型的约束，包括基本约束、参与约束、并发约束、特殊约束和一般约束等。

为了以后叙述方便，在本节中，首先简单介绍一下参与约束、并发约束和基本约束。

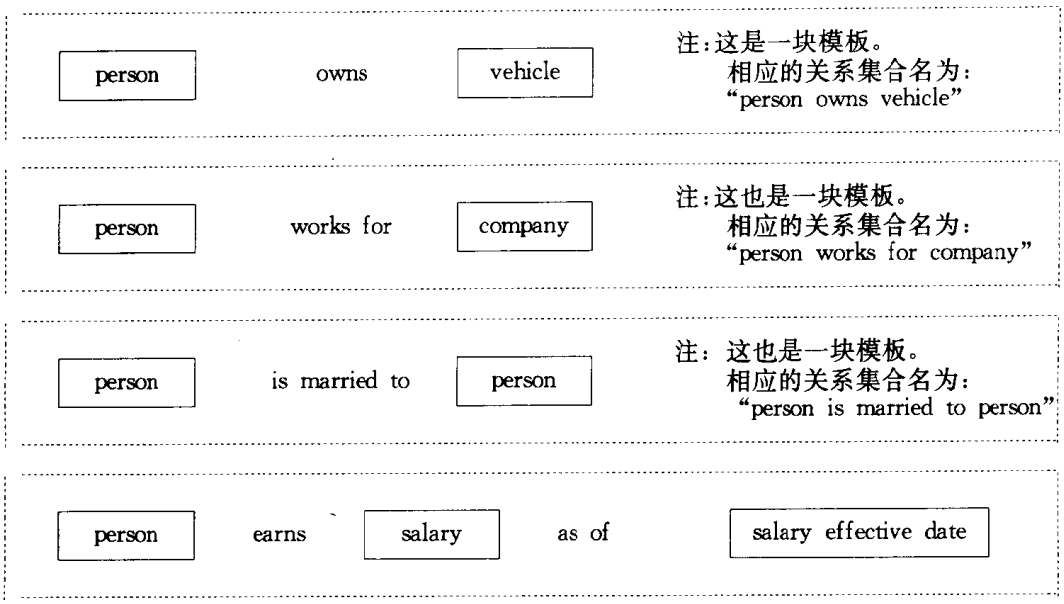


图8.4 关系集合模板

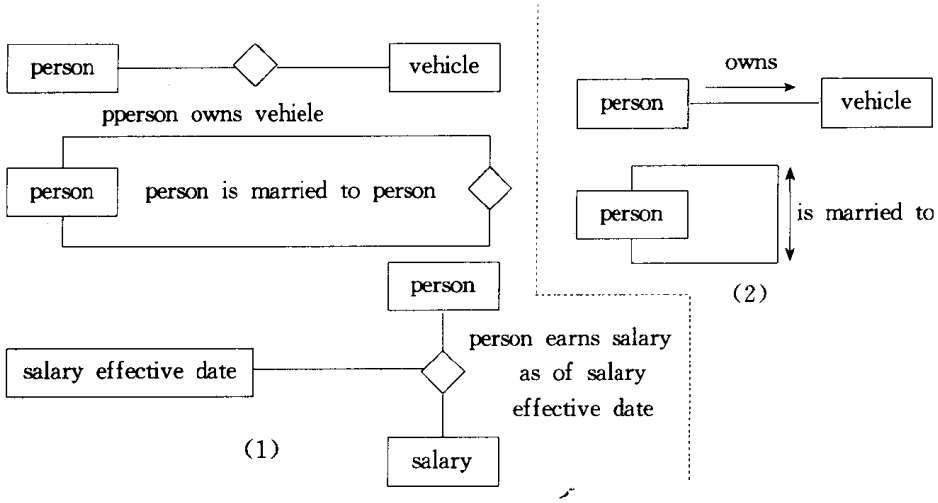


图8.5 关系集合图例

(1) 参与约束(participation constraints)

给定一个关系集合,参与约束定义了对象类中的一个对象可以参与该关系集合中的关系数目。例如,图8.6是一个关系集合。

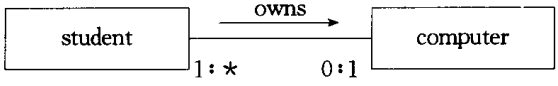


图8.6 参与约束示例

如果把这一关系集合"展开"为如图8.7形式,就可以很容易理解该参与约束的含义。其中,"1:*"和"0:1"均是参与约束。"1:*"表明一名学生可以有一台或多台计算机,而"0:1"表明一台计算机最多能够属于一名学生。

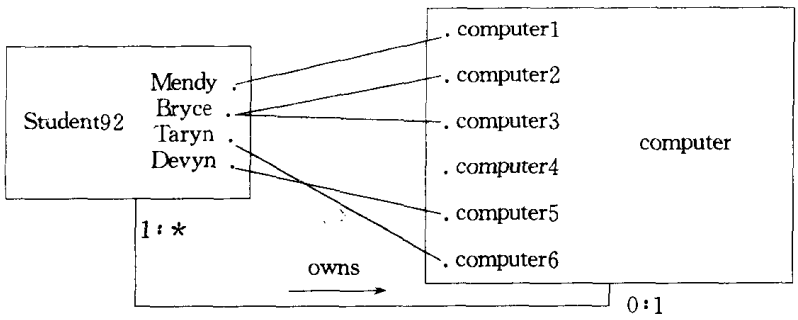


图8.7 参与约束的解释

在OSA中，参与约束的基本形式是min:max。其中，min是非负整数，它表示该类的一个对象可参与的最小的关系数目；max是非负整数或为"*"，"*"表示一个大于min的非负整数，max指出该类的一个对象可参入的最大的关系数目。如果min和max相等，那么该参与约束可写为min或max。参与约束应写在对象类的连接附近。

(2) 并发约束(co-occurrence constraints)

对于多元关系而言，参与约束有时还不能很好地表达我们的一些要求或限制，为此，OSA引入了并发约束。

在一个多元关系集合中，并发约束定义了一个对象类中有多少个不同对象可以与其它对象类中的特定的一个对象或特定的一组对象一起出现。如图8.8所示。

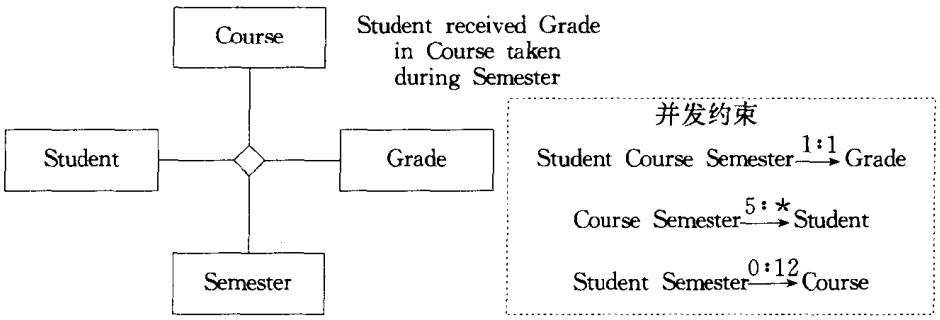


图8.8 并发约束示例

其中，"Student Semester $\xrightarrow{1:12}$ Course"表明，对于Course对象类，可以与特定的那组对象"Student Semester"同时出现的不同对象数目最小为1，最大为12。就是说，一名学生在一学期中，最少可以学习一门课程，最多可以学习12门课程。

在使用并发约束时，应注意关系集合的有效性。

(3) 基本约束(Cardinality Constraints)

基本约束定义了一个对象类中对象的数目。如图8.9所示。

由图8.9可知，基本约束的形式与参与约束的形式一样，只不过基本约束要写在矩形的右上角。当没有给出基本约束时，则意味着该约束为0:*。

(4) 一般约束(General Connstraints)

尽管以上给出的约束可以表达有关ORM的一些限制，但是，在分析中，它们还不可能满

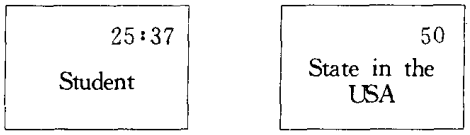


图8.9 基约束示例

足分析员的需要，为此，OSA 引入了一般约束。

一般约束是一陈述语句，用于进一步限制 ORM 图中的对象类和关系集合。分析员可以按着自己希望的形式给出一般约束，例如，谓词演算符号或自然语言等。

8.1.2 特殊的关系集合

在建造系统模型中，人们经常使用“抽象”等构造方法。为此，OSA 引入了三种特殊的关系集合：“is a”关系集合、“part of”关系集合以及“is member of”关系集合。在本章中，为了叙述方便，一律把“关系集合”简称为“关系”。

1. 一般——“is a”关系

“is a”关系指出，一个对象类中的每一对象是另一对象类的一个对象。

如果对象类 B 中的每一对象是对象类 A 中的对象，那么，我们把 A 称为超类或称为一般类；相对地，把 B 称为子集或称为特殊类。通常，“is a”关系表达了继承。

在构造系统中，由于“is a”关系是我们经常使用的构造方法，因此 OSA 引入特殊的符号来表述这一关系，即用空心三角形以及和一般类、特殊类相连的线段来表示。如图8.10(1)所示。

当一般类与多个特殊类具有“is a”关系时，可以把这一情况以图8.10(2)的形式给出。

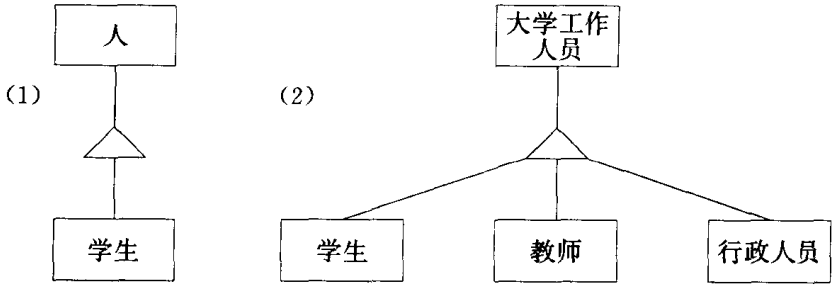


图8.10 “一般/特殊”关系示例

2. 聚合——“is part of”关系

“is part of”关系表明：一个对象，称之为聚合，是由一些称之为成分的对象构成的。在构造对象关系模型图中，我们常常涉及的“组装”关系、“容纳”关系、“包含”关系等，都是“is part of”关系，因此，这种关系也是我们经常使用的一种构造方法。

由于可以使用不同的方法把一个聚合分为若干个成分，因此对于一个聚合对象类而言，可以存在多个“is part of”关系。于是，“is part of”关系的一般表示如图8.11所示。

在“is part of”关系中，可以使用参与约束。考虑到各成分对象参与各自关系的数目可能有所差异，因此，如果采用图8.10(2)的形式表达聚合，那么对聚合的参与约束就要写在实心三角形的下面，如图8.11所示。

3. 联合——“is member of”关系

“is member of”关系用于生成一个由对象构成的集合，并把该集合看作是一个对象。

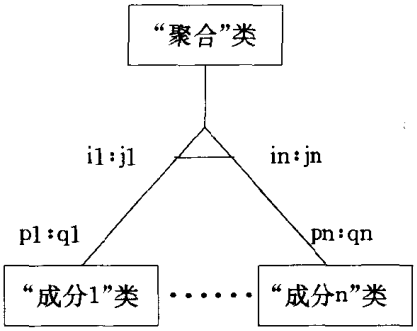


图8.11 "一般/特殊"关系

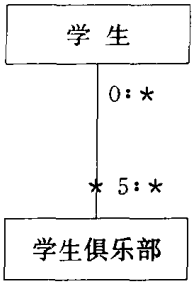


图8.12 "联合"示例

如果一个对象类，它的对象是集合，我们把这样的对象类称为集合类或联合(association)。如果联合的每一对象是由某一对象类中的对象构成的，那么，我们把这一对象类称为成员类或全体(universe)。"联合"意指一些成员联合在一起构成了一个对象；"全体"意指生成子集的那个对象的集合。如图8.12所示。其中，"学生俱乐部"是集合类(联合)，"学生"是成员类(全体)。每一俱乐部是一个联合，是成员类的一个子集。

在OSA中，用一条带"*"的线段表示"is member of"关系，其中，要把"*"写在联合那一端。如图8.12所示。这一关系的读法，是从"成员类"读向"联合"。就图8.12给出的例子而言，应读为"学生是学生俱乐部的成员"。

参与约束可以用于"is member of"关系。在图8.12中，参与约束"0:*"表明，一名学生不一定是任一俱乐部的成员，然而，每一学生可能是多个俱乐部的成员。"5:*"表明，每一学生俱乐部必须至少有5名学生。

"is member of"关系总是二元关系。

8.1.3 特殊对象类、资格条件、注释

1. 特殊对象类

在OSA中，有两种特殊对象类。一种是"单一对象类"，即在该对象类中只有一个对象；一种是"关系对象类"，即在该对象类中，每一对象均是一个关系。

引入单一对象类，主要是为了解决概念上的问题。由于OSA只定义了关系和关系集合，而没有定义对象与对象类的连接，因此引入单一对象类，便很容易地解决了这一问题。

2. 资格条件

在构造ORM中，为了确定一个对象是否属于某一对象类，为了确定一个关系是否属于某一关系集合，OSA引入了"资格条件"这一概念。

对于某一对象类或某一关系集合，所谓资格条件就是对该对象类或该关系集合施加的那些约束的交。如果一个对象满足某一对象类的资格条件，那么它就是该对象类的一个成员。如果一个关系满足某一关系集合的资格条件，那么它就是该关系集合的一个成员。

3. 注释

OSA允许分析员为ORM添加注释，包括图示、解释等，以便使ORM具有更多的信息，使之更易理解。但是，添加的注释不能形成对对象类、关系集合的限制。

8.1.4 对象关系模型小结

为了捕获一个系统的说明性信息，目前的分析技术以及CASE工具正摆脱传统的数据结构模型，逐渐采用源于ER模型的语义数据模型，引入了“一般”、“特殊”、“聚合”和“联合”等抽象机制。OSA的ORM也是一种语义数据模型，并用ORM图表示之(如图8.13所示)。

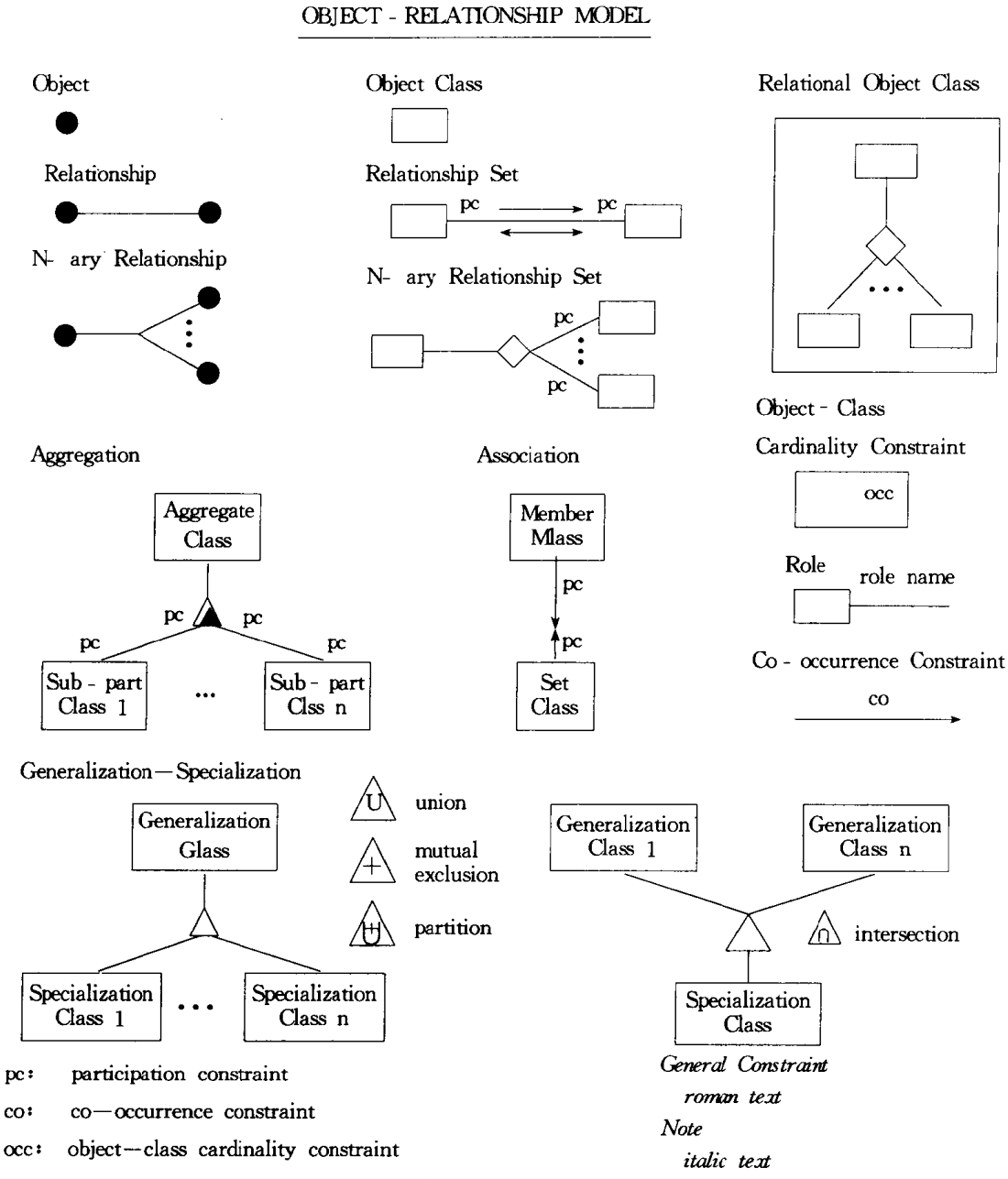


图8.13 ORM图

为了构造ORM图，OSA给出了五个基本概念，即对象、关系、对象类、关系集合和约束。其中，对象类是OSA的基础，刻画关系集合的图是OSA对象关系模型的基本成分。

在OSA中，不把属性作为ORM的基本构造，并认为在分析阶段说明属性是超前活动，有

着潜在的危害。其理由是：

·在分析过程中，很难定义属性，因为这一概念涉猎过广；并且很难区分对象和属性。

·在分析过程中，若标识属性，会导致不必要的复杂构造，例如，重值属性、复合属性以及弱实体等；还可能表达了不该表达的属性间关系，引起以后数据规范化和模型集成中的一系列问题。

OSA 认为，在设计阶段，可以使用类似综合数据库系统的过程，自动地定义属性。

8.2 对象行为模型

对象行为模型用于描述系统中各对象的动态结构，即记录可察觉的对象状态、从一种状态转换为另一种状态的条件和事件以及对象执行的动作和对它所施行的动作。OSA 的对象行为模型是用状态网(state nets)表示的。

为了构造对象行为模型，OSA 集中于三个基本概念：状态(state)、触发(trigger)和动作(action)。这三个概念的模型化是状态网的基本构造。

8.2.1 基本概念及概念模型化

1. 基本概念

(1) 状态

在 OSA 中，一个状态表达了一个对象的外征(status)、阶段(phase)或活动(activity)。如图8.14所示。

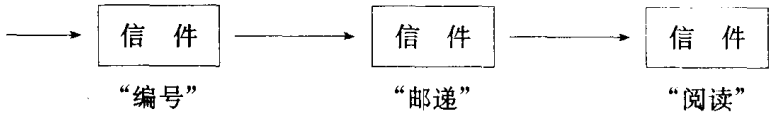


图8.14 “信件”的状态

其中，“编号”、“邮递”、“阅读”均是信件的不同状态。这组状态是可观察到的，它们刻画了信件的变化。并且，还表达了一类“信件”变化所具有的共性，因而 OSA 为每类对象建立相应的对象行为模型。其中，对象在不同阶段所呈现的各种状态的集合，构成了行为模型化的基本部分。

我们可以用许多方法获得对象状态信息。一般，通过对系统中对象的观察或想象，可以得到该对象应具有的不同状态。

(2) 触发与转换

尽管标识对象状态是行为模型化的主要部分，但是，如果不了解一个状态与另一状态的联系，它也是没有多大用处的。因此，必须知道促使状态变化的事件和系统条件。

对象状态变化的过程称为转换。促使状态转换的事件和条件称为触发。如图8.15所示。其中，“⟶”表示转换，“⟶”之上的“封装与投递”等描述了触发。对象“信件”响应这一触发，从“编号”状态转换为“邮递”状态。

(3) 动作

为了建立对象行为模型，不但要把状态转换模型化，还要把动作模型化。

在 OSA 中，把动作分为两类：不可中断动作和可中断动作。所谓不可中断动作是指分析

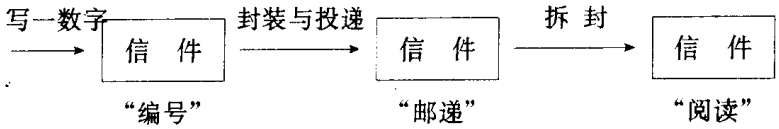


图8.15 状态转换的触发

员期望的那些一直执行到完成的动作，除非系统运行失败。所谓可中断动作是指那些在其执行中可以被挂起，以后还可以重新执行的动作。OSA 把不可中断的动作视为与状态转换有关的动作，而把可中断的动作视为与状态有关的动作，并把可中断动作模型化为一个离散的、运行时间长的活动(activity)，该活动可以被中断，以响应作用于对象的事件或条件。例如，图8.15中的“封装与投递”就是一种不可中断的动作，而“阅读”状态中，阅读信件就是一种可中断的动作。

2. 概念的模型化

(1) 状态的模型化

在OSA 的状态网中，用圆角矩形表示对象状态。每一这样的矩形对应对象的一种状态，其中给出表示该状态的状态名。例如，假定我们有一台机器人叉车，它的职能是把粮库中的粮袋装入运粮卡车。为了简化，我们假设该叉车有以下四种状态——“呆闲”、“驶向粮库”、“搬运粮袋”和“驶向卡车”，那么该叉车的这组状态如图8.16所示。

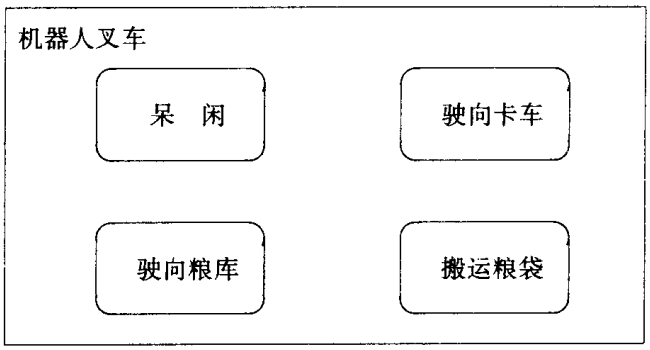


图8.16 状态模型化示例

为了标识特定对象类的一组状态，用表示类的矩形把该组状态框起，并在矩形的左上角，给出该对象类的名字。

相对于特定的对象类，任一时刻，一种状态只有两种值：on 或 off。当对象正处于该状态时，其值为 on，否则为 off。

(2) 转换的模型化

在OSA 的对象行为模型中，用矩形表示触发和动作。其中，把矩形分为两部分。上半部分给出触发的描述，即给出条件(当该条件满足时，引起一个状态转换)。下半部分描述在状态转换期间所发生的动作。如图8.17所示。

由该例可以看出，其中的动作可由一个或多个操作组成，这些操作在状态转换的某一时刻可能执行了，也可能没有执行，即是说，动作在概念上有多条执行路线。但是，只有所有路线均执行完成时，才实现了这一转换。

图8.17中触发“到达粮库”前的符号“@”指出该触发是基于事件的。在OSA 中，把这一符

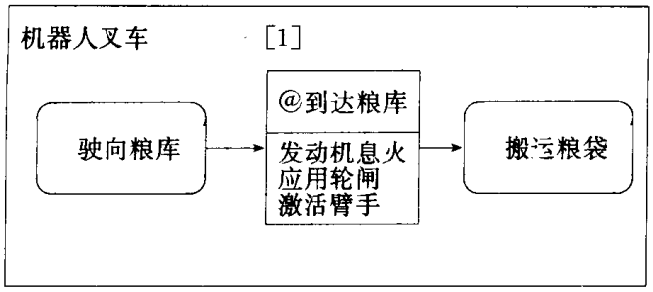


图8.17　转换模型化示例

号读为“基于”、“当”等。就图8.17给出的例子而言，可以读为“当到达粮库”。

在转换框左上角给出的“[1]”为转换标识符，以便以后引用这一转换。

在转换之前的状态称为“前状态”，而称转换后的状态为“后状态”。当一个对象处于某一转换的前状态时，我们称这一前状态为该转换的就绪状态。当一个转换处于就绪状态时，相应的触发才能使该转换发生。在完成了转换之后，对象处于后状态。

3. 触发条件和事件

正如以上所述，当指定的系统事件发生或当确定的系统条件满足时，一个触发导引了一个对象的转换发生。相对于一个对象而言，这些事件和条件可能是外部的，也可能是内部的；甚至可能是一些事件和条件的组合构成了一个触发。一般而言，一个触发是一个产生真假值的布尔表达式。

(1) 基于条件的触发

条件是有关对象当前状态、系统环境当前状态、对象存在不存在、对象之间关系存在不存在的逻辑陈述。在 OSA 中，可以使用形式的或非形式的逻辑陈述，作为触发的条件。当条件为真时，相应的触发引起就绪的转换发生。

(2) 基于事件的触发

一个事件是系统的任一变化。例如，创建一个对象、删除一个对象、一个对象的状态转换、开始一个活动、接收一个命令、发送或接收一条消息等。任一可被察觉的变化均可被模型化为一个事件。因此，对象响应一个事件相当于该对象响应系统的一个可被察觉的变化。

在 OSA 的行为模型中，把一个触发中发现事件的那一部分称为事件监视器。事件监视器是概念上的设备，它能够发现一定类型的系统事件。事件监视器的名字是由相对应的触发和该触发所监视的事件类型合在一起而构成的。例如，图8.17中的“@到达粮库”就是事件监视器的名字。

事件与条件之间的区别是：事件仅在发生时触发了一个就绪的转换，而条件在该条件满足的整个期间触发就绪的转换。

在某些特定的情况下，为了记录事件及与该事件相关的信息，OSA 允许把事件作为对象，把一些类似的事件作为一个对象类，也允许对事件对象类添加关系的参与约束。

(3) 复合触发

在触发描述中，把条件和事件监视器一起使用，便形成了复合触发。如图8.18所示。

其中，OSA 使用“∧”和“∨”，分别表示布尔“与”和布尔“或”。对此，只要适于相应的问题域和读者。OSA 允许使用其它等价的符号，由此可见，触发可以是事件和条件的“与”和“或”。

精确地说，任一时刻，系统中不可能有两个事件同时发生，因此，在构造复合触发时，若出

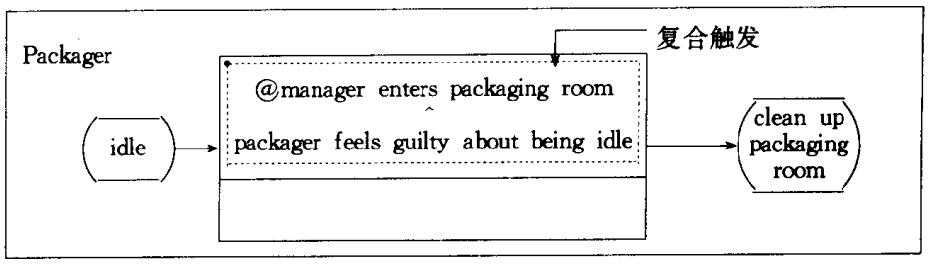


图8.18 复合触发

现两个事件的“与”，就应当认真地考虑，看它是否符合实际系统的行为，千万不能为了缩短行为模型化过程而简单地使用复合触发。

8.2.2 状态网

前面已经提及，OSA 用状态网来表示对象行为。构造状态网的主要成分是模型化的状态和转换，因此，状态网是一种符号结构，表示对象类中所有对象的状态和状态转换。状态网可以被看作是行为模板，指出一个对象类中的每一实例具有该模板所表示的行为。

当分析员发现系统中存在类似的行为时，可以使用状态网来描述之。如果把一个状态网与一个对象类联系起来，则说明了该对象类的每一实例具有该状态网所描述的行为。OSA 给出了如下几种结构的状态网。

1. 后状态(Subsequent States)

后状态是转换之后的状态。在状态网中，用箭头线的箭头指向转换的后状态，而箭头线的尾和表示转换的矩形相连。

转换可以有不同形式的后状态。在图8.19中，给出了三种基本形式。

图8.19(1)表明，当转换的动作 b 完成后，状态 s1变为 on，即对象处于这一状态。

图8.19(2)表明，当转换的动作 b 完成后，状态 s1，s2变为 on，即对象同时执行这两种状态所对应的活动。

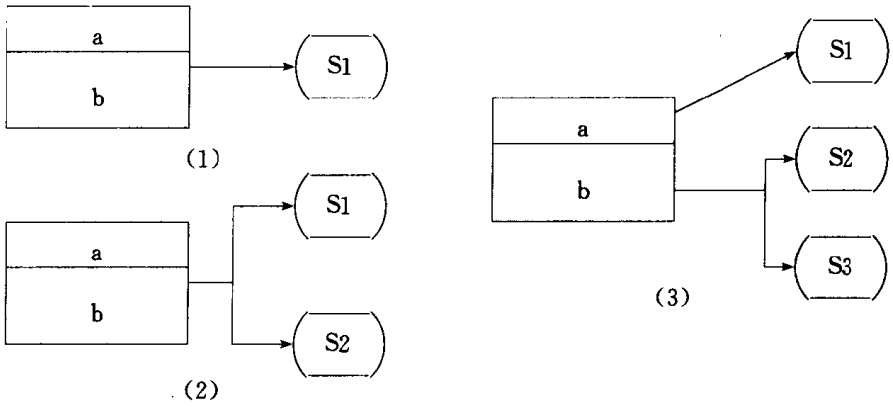


图8.19 转换的后状态

在图8.19(3)中，由于该转换有多条箭头线，分别指向多个后状态，因此存在一个选择问题，并且只能选择其中一条予以使用。即当转换完成时，对象或处于 s1状态，或处于 s2，s3状

态。对于后状态的不确定性，可以使用约束，限制后状态的选择。如图8.20所示。

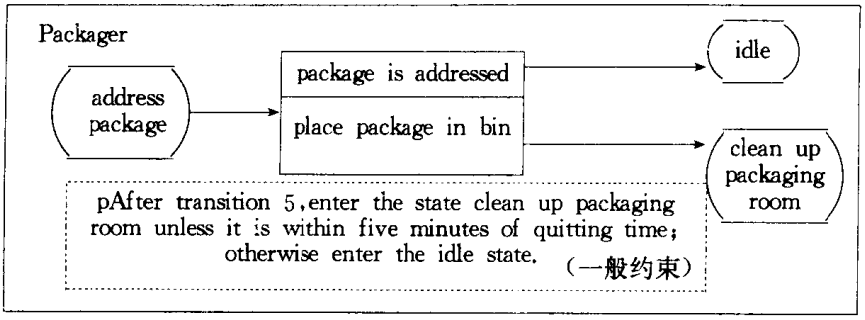


图8.20 带约束的后 态

2. 前状态(Prior States)

前状态是一个转换之前的状态。在状态网中，用箭头线的箭头指向转换，箭头线的尾与前状态相连。和后状态类似，也存在不同形式的前状态。图8.21给出了三种基本形式：

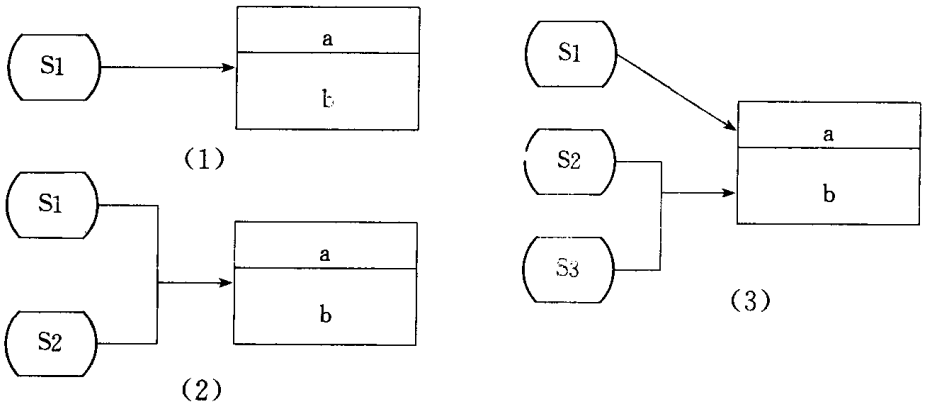


图8.21 转换的前状态

图8.21(1)表明：当状态s1有值为"on"时，该转换是就绪的，即在这种情况下，若触发a发生，则可以使该对象从s1状态转向相应的后状态。

图8.21(2)表明：当状态s1和状态s2均有值为"on"时，才能使该转换成为就绪的。

图8.21(3)表明：状态s1为"on"，或状态s2，s3为"on"，或状态s1，s2，s3为"on"，该状态是就绪的。当三个状态均为"on"时，依然存在一个选择问题。如果没有给出约束，则该选择是随机的。当触发发生时，使选取的状态变为off。

3. 初始转换(Initial Transition)

当对象最初进入系统时所呈现的状态，称为该对象的初始状态。初始转换激活对象的初始状态。显然，初始转换没有相应的前状态，它总是就绪的，即不管它的触发是否被满足。完整的状态网必须给出一个初始转换。

"@create"一般用作察觉新对象产生的事件监视器。许多初始转换使用这一事件触发器，当然，也可以使用其它与之等价的事件触发器，例如："@birth"，"@hire"等。图8.22给出了两个等价的状态图，都初始化了一个对象"雇员"。

其中，给出的初始转换描述了雇员的创建，"idle"是雇员的初始状态。

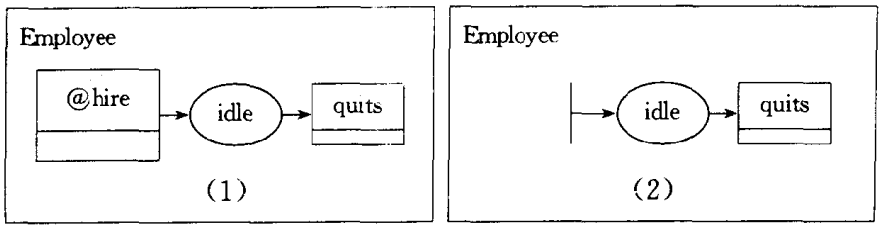


图8.22 初始与最终的状态转换

如果在初始转换中没有转换动作，则可以把这一初始转换简写为一个竖杠，如图8.22(2)所示。即竖杠表示着没有初始化动作的初始转换，其触发为“@create”或其它等价的事件监视器。

如果分析员需要说明初始化动作，那么就不应该使用这样的简写形式，而应采用图8.22(1)所示的形式，并在转换的动作部分给出初始化动作的描述。

4. 最终转换(Final Transition)

最终转换是没有后状态的转换。当最终转换发生时，其前状态变为“off”。如果一个对象的所有状态均为“off”，则意味着该对象结束其生存，在系统中消失了。

最终转换是可选的。若一个对象有离开系统的要求，才有必要选用最终转换。与初始转换类似，最终转换的事件监视器为“@destroy”或其它等价的事件监视器，它触发了转换中的动作，导致对象的终止。在图8.23中，给出了最终转换的两种表示形式。

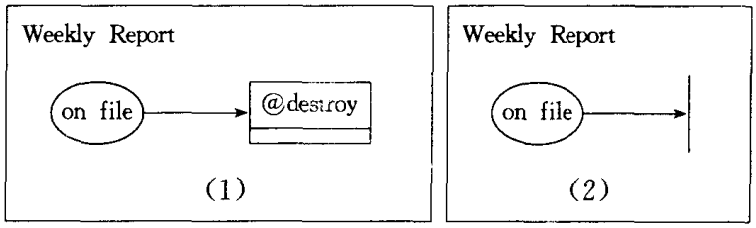


图8.23 最终转换的简写符号

图8.23中，(2)是(1)的简写形式。竖杠“|”意指该转换是“@destroy”或其它等价的没有转换动作的事件监视器。与初始转换一样，如果需要描述最终转换的动作，那么就应当采用(1)的形式。

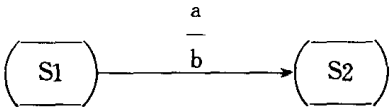
5. 转换的缩写

大多数转换具有单一入口和单一出口，对于这样的转换，OSA 给出了相应的简写形式。如图8.24所示。

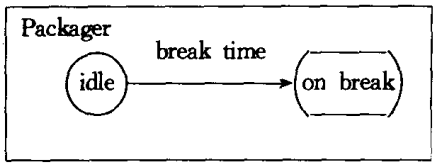
在图8.24(1)中，给出了状态转换的一般简写形式，即在横杠“—”上给出触发，在横杠下给出转换动作。

图8.24(2)表明：如果一个转换没有转换动作，那么可以只给出触发，并把该触发写在箭头线之上。如果状态转换不需要什么事件或条件 也不需要转换动作，那么可以用一条箭头线直接把该转换的前状态和后状态连接起来。图8.24(3)就是这样的一种情况。它表明只要“on break”状态为“on”，“clerk”便进入“ready to work on orders”状态。

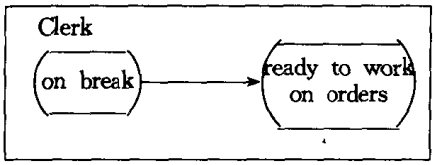
除了以上的简写形式外，还有一种转换的简写形式，即在状态网中使用了双箭头线。双箭



(1):状态转换的简写符号



(2):"无动作"转换的简写符号



(3):"无动作"且"具有隐含事件"
转换的简写符号

图8.24 转换的简写符号

头线意味着:处于某一状态的对象,当响应了一个触发,在执行完该转换动作后,还需要返回那个状态。图8.25表示了这种情况的简写形式。有时,为了使语义更加确切,往往需要添加一般约束。

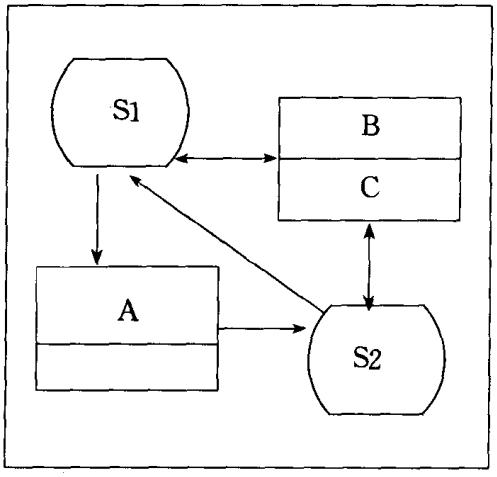


图8.25 状态网中的双箭头线

6. 状态的保留

OSA 允许一个对象进入新状态后,并不离开以前的状态,使该对象同时处于两个状态之中,保留了原来的状态,使之依然为"on"。在图8.26中,给出了状态保留的示例。

图中在状态"driving to work"右边的")"表示了对这一状态的保留。

7. 例外(Exception)

例外是一种系统事件或条件,它不是系统正常行为的一部分。如图8.27所示。图中箭头线上的竖线"|"表示例外。

图8.27(1)表示,触发"@paper jam"是一例外。

在状态的转换期间也可能发生例外,即转换动作的例外。在这种情况下,要把竖线画在转

换框附近，并给出例外的描述。如图8.27(2)所示。

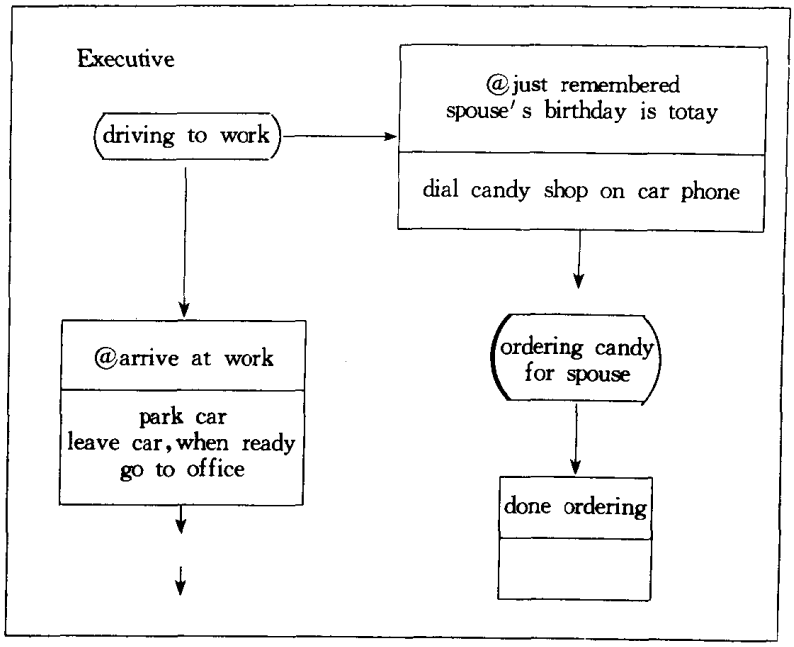
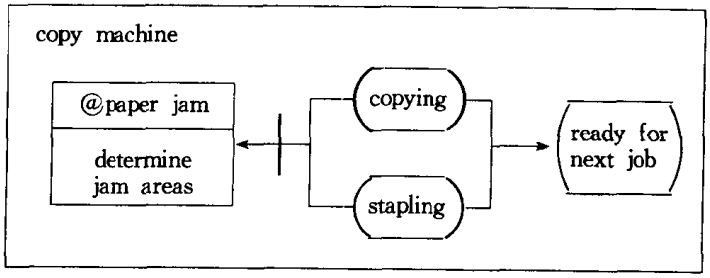
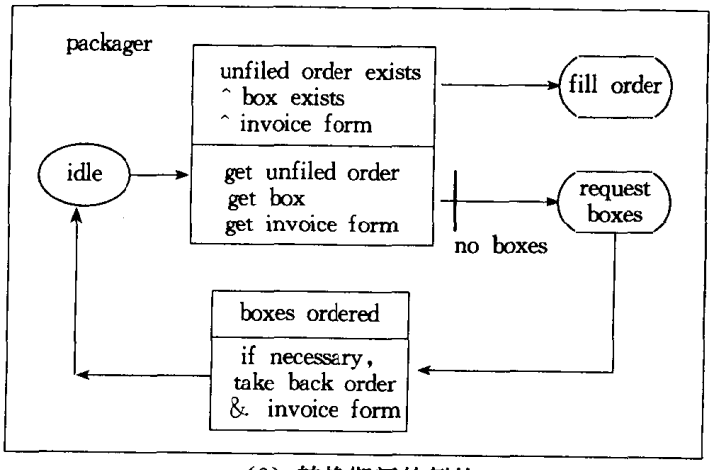


图8.26　不改变前状态值的转换



(1) 例外的模型化



(2) 转换期间的例外

图8.27　例外

在状态网中，给出例外或不给出例外，它所描述的行为是一样的。给出例外，其好处是，可以使分析员指定一条特殊的路径，作为正常行为的例外，以便施行有效的控制。

8. 实时约束(real-time constraints)

当时间是行为描述的一个重要成分时，分析员可以在状态网中给出有关时间的约束，以说明时间上的需求。OSA 允许把时间约束应用于“触发”、“活动”、“状态”以及“状态转换路径”。图8.28中的“{…}”就是一些时间约束。

在图8.28(1)中，状态“on break”附近的约束“{≤15 minutes}”表明，“packager”处于“on break”状态最多只能15分钟。即限制了对象“packager”在状态“on break”中的持续时间。

图8.28(2)中的约束“{≤30 seconds}”限制了转换活动的最大持续时间。

图8.28(3)中的约束指出：当经理进入包装车间时，在“idle”状态中的“packager”必须在两秒钟内进入下一状态。

图8.28(4)中的约束给出了整个转换的时间限制。

图8.28(5)中，使用了路径标识“{a}”，“{b}”，时间约束“{a to b≤10 minutes}”表示，在“{a}”和“{b}”标识的路径内，其转换和活动最多使用10分钟。

OSA 没有限定时间约束的表示格式，在具体的系统分析中，可以采取灵活自由的格式。一般地，当初始的对象行为模型建立以后，就可以给出时间约束。

9. 状态网的一般/特殊

在实际系统中，不但对象类存在着“一般/特殊”关系，而且在行为描述方面也存在着“一般/特殊”问题。

特殊类的对象不仅表现了该类的共同行为，还表现了相应一般类的行为，即特殊对象的行为包含着一般对象的行为。

OSA 允许从两个不同方向开发状态网。当我们把一个状态网特殊化时，可以使其中的触发和动作具有更多的特性，增加一些特殊的行为。从而，可以认为，特殊类的状态网是由一般类的状态网生成的。反之，我们也可以对给定的一组状态网，通过标识和提取共同行为，生成一般的状态网。从而，也可以认为，一般类的状态网是由特殊类的状态网生成的。

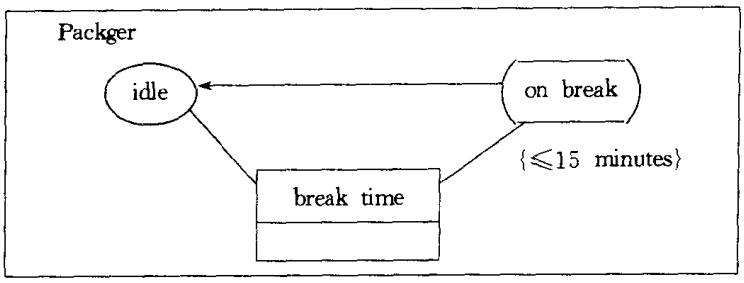
当特殊类和一般类之间的行为，相对于状态和转换而言没有多大区别，或这一区别可以标识时，则不必重新把整个特殊类的状态网画出，可以只画出那些与一般类不同的行为，形成特殊类简化的状态网。这一简化有以下两点益处：

① 由于简化的状态网，仅给出特殊类与一般类状态网的不同，因而可以减少潜在的错误；

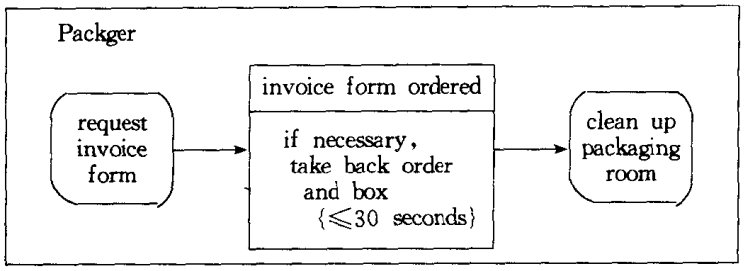
② 使用简化的状态网，容易了解特殊行为与一般行为的不同。

如果对象类的状态网是以简化的方式给出，那么，在概念上就相当于对“一般/特殊”层次结构中的所有一般类增加了细节。

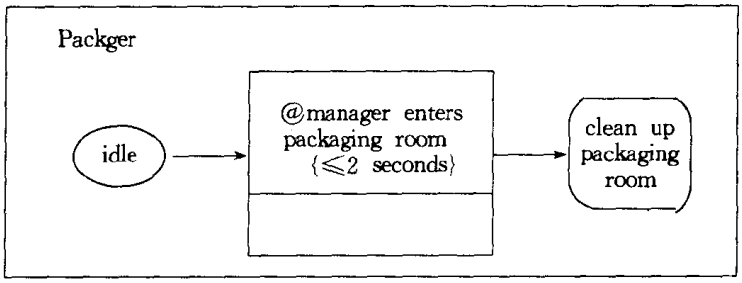
复杂的状态网(例如，多层次、多继承)，使用简化形式，会在概念上产生控制上的困难。因此，在这种情况下，OSA 建议使用标准的状态网。



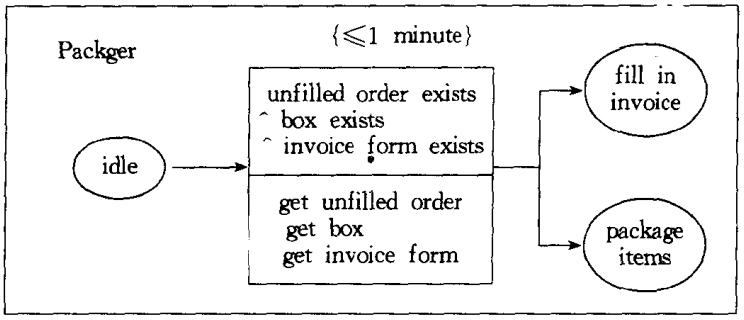
(1) 状态持续时间的约束



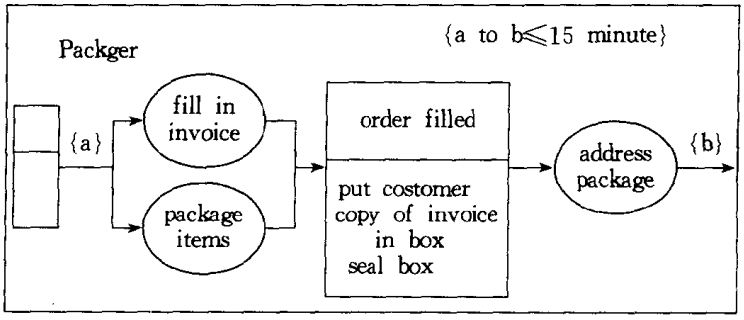
(2) 转换中动作的时间约束



(3) 触发的时间约束



(4) 整个触发的时间约束



(5) 状态网路径的时间约束

图8.28 实时约束

8.2.3 对象行为模型小结

为了捕获行为信息，目前的分析技术一般不大采用整个的系统模型，像操作模型、过程相互作用模型以及有限状态机等，而使用面向对象的行为模型。

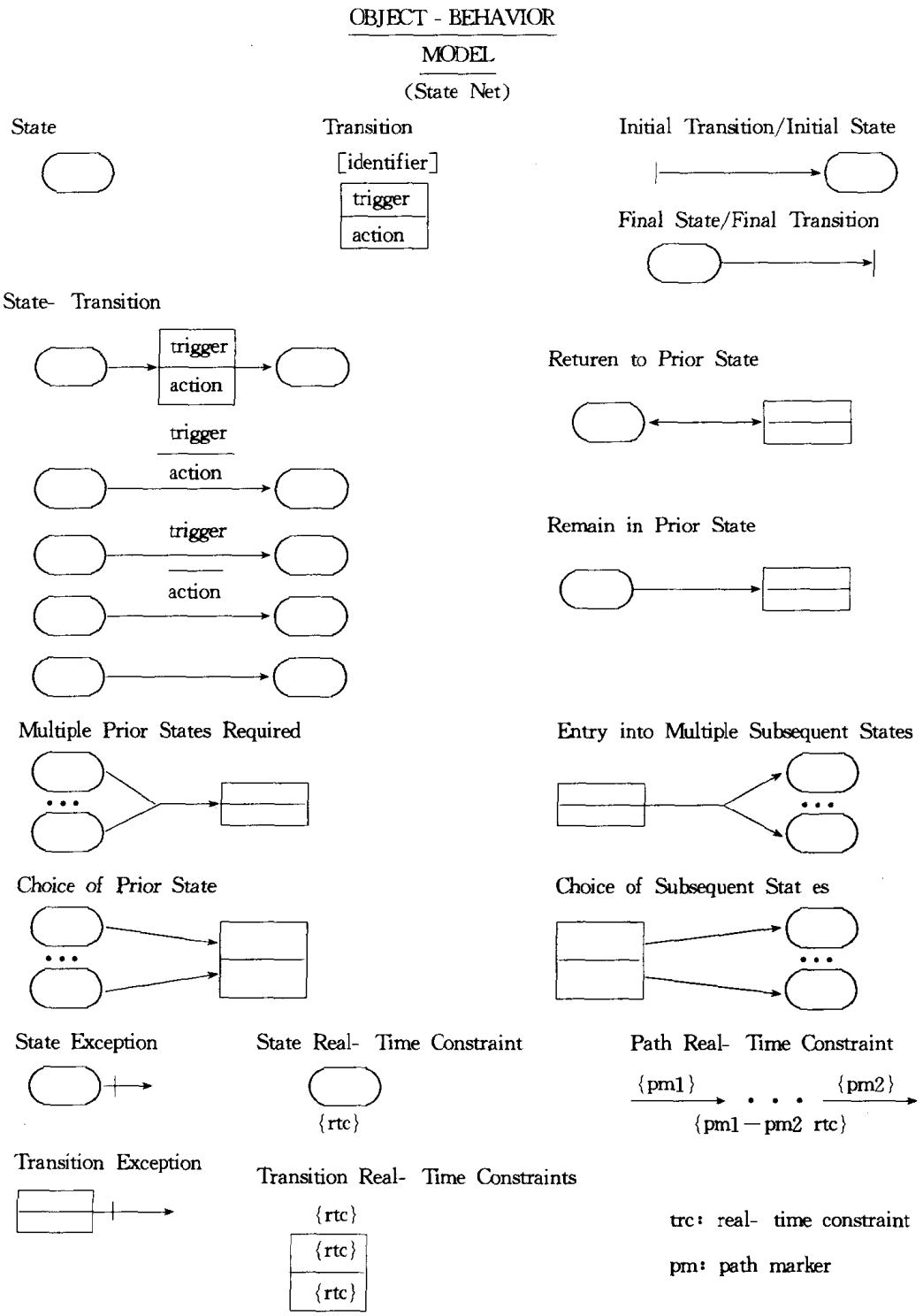


图8.29 OSA对象行为模型中所用的表示符号

OSA 的对象行为模型是一种面向对象的行为模型，并且用状态网表示之，它描述了对象类中所有对象的共同行为。

在构造对象的行为模型中，OSA 围绕着三个基本概念：状态、触发、动作，来组织对象的行为信息。这三个概念的模型化是状态网的主要成分，其中，对象的状态集合是状态网的基本框架。

支持并发是 OSA 对象行为模型的突出特征。由于它是面向对象的，因此可以表达不同对象类的对象并发；由于一个状态网可以被看作为一类对象的行为模板，因此可以表达同一对象类中不同对象的并发；又由于 OSA 提供了多重后状态、多重前状态以及状态保留等机制，于是可以表达一个特定对象不同动作的并发。

另外，OSA 还支持具有时间延迟的转换，支持例外，支持整个对象的“一般/特殊”抽象等，这一切弥补了模型驱动分析技术在表示能力上的一些不足，形成了自己独特的风格。

图8.29是 OSA 对象行为模型中所用的表示符号。

8.3 对象交互模型

分析员可以使用对象关系模型来描述对象之间的关系，可以使用对象行为模型来描述各类对象的行为，但是，为了宏观了解系统行为及功能，就必须刻画对象之间的相互作用。

刻画对象之间的相互作用，就是描述以下三方面的内容：

① 在对象交互中，涉及了哪些对象；

② 在对象交互中，每一对象是如何活动的；

③ 对象交互的本质是什么。

由8.1，8.2节可知，对象关系模型描述了第一方面的内容，对象行为模型则描述了第二方面的内容。OSA 引入一种新的机制来描述交互的本质，即描述交互的行为以及交互中被交换的信息。根据这一机制，以及相关的 ORM 和状态网，可以构造 OSA 对象交互模型。

8.3.1 基本的对象交互

在 OSA 中，把交互的基本元素分别称为：起始对象、终点对象以及交互链。如图8.30所示。

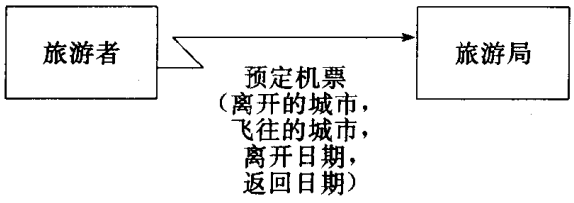


图8.30 交互描述示例

其中，“旅游者”为起始对象，“旅游局”为终点对象，“Z 形箭头线”为交互链。并把“预订飞机票(飞往的城市……)”称为交互标识。图8.30构成了一个简单的交互描述。它描述了该交互的行为和交互中被交换的信息。

OSA 提供了以下四种基本描述交互的格式，如图8.31所示。

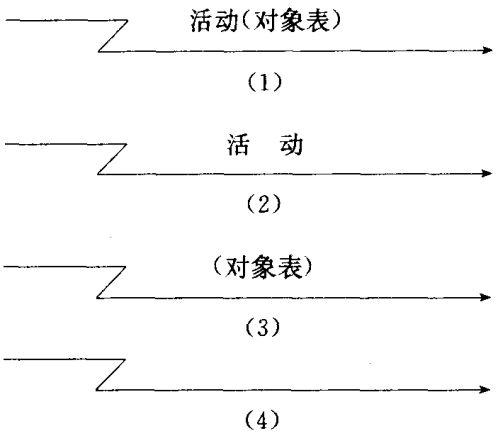


图8.31 描述交互的基本格式

图8.31中,图8.31(1)表示:在这一交互中,既有交互的行为,又有被交换的信息;图8.32(2)表示:在这一交互中,只有交互的行为;图8..31(3)表示:在这一交互中,只有被交换的信息。这样的交互通常被称为通信,即一个对象向另一对象发送一条消息,其中动作的默认值为通信或其它可能的同义词。由于消息可能是可感知的,或是不能被感知的,因而,把消息放在括号内。

在图8.31(4)所示的交互中,没有给出交互标识。它表明该交互或不言自明,或分析员不能用简单的交互标识清晰地描述之。

8.3.2 特殊类型交互的描述

1. 对象间的同步交互

在实际生活中,存在对象间的同步交互。例如,在我们正常通信中,当发送者发送一条消息时,接收者必须做好接收这一消息的准备。如果接收者没有做好接收这一消息的准备,则该消息自然丢失。因此,为了成功的通信,发送者和接收者必需了解通信发生的条件。据此,OSA 借助于状态网,描述对象间的同步交互。如图8.32所示。

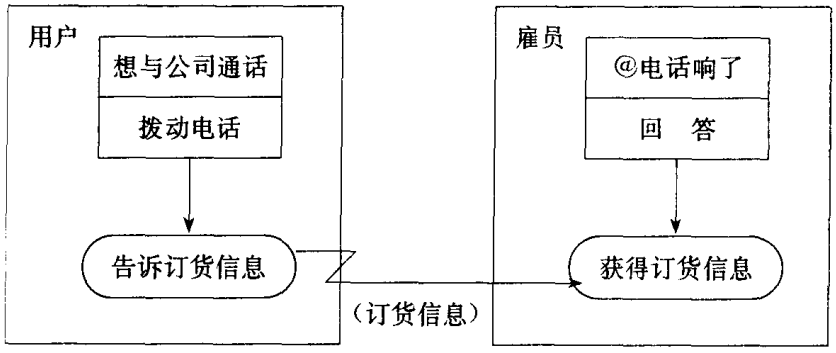


图8.32 同步交互示例

2. 对象间的异步交互

对象间的异步交互在实际中是经常发生的。我们给自己的亲人或朋友寄信,就是对象间的异步交互的例子。

在OSA 中，通过提供“中介”对象（例如，邮局），建立对象间的异步交互。如图8.33所示。

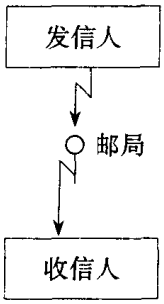


图8.33　异步交互示例

3. *说明特定的交互对象*

在以上的例子中，均没有指出特定的交互对象。但在实际中，常常需要指出参与交互的特定对象。为此，OSA 引入相应的符号，描述这一类型的交互。如图8.34所示。

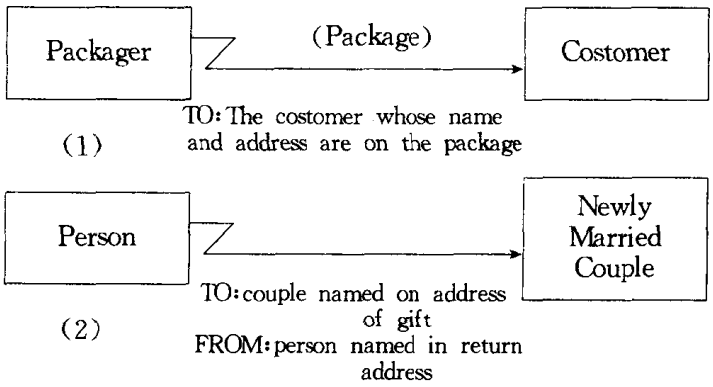


图8.34　特定对象间的交互

其中，“TO：…”为标识信息，用于指出一个对象类中的特定对象。即 OSA 使用了“TO——短语”来标识交互中特定的终点对象；并使用了“FROM——短语”来标识交互中特定的起始对象。

4. *与多个对象的交互*

在实际生活中，还存在涉及多个起始对象和多个终点对象的交互。例如，新闻广播就是这样一类的交互。

在OSA 中，使用如下符号，描述这一类型的交互。如图8.35所示。

图8.35中，图8.35(1)表明：一个对象与多个对象发生交互；图8.35(2)表明：一个对象与一个对象类中的多个对象发生交互；图8.35(3)表明：一个对象类中的多个对象与另一个对象类中的多个对象发生交互。在这一抽象级上，我们可以认为类 clerk 中的所有雇员一起工作，布置屋内所有设备；图8.35(4)表明：公司经理向职员和工人发布消息。

5. *双向交互*

有时，一对交互是紧密相关的。在这种情况下，OSA 允许把这样的一对交互看作是同一交互，因此，可以把它简写，如图8.36(1)所示。

如果双向交互两边的描述是一样的，则只需在 Z 形附近给出一次交互描述。如图8.36(2)所示。

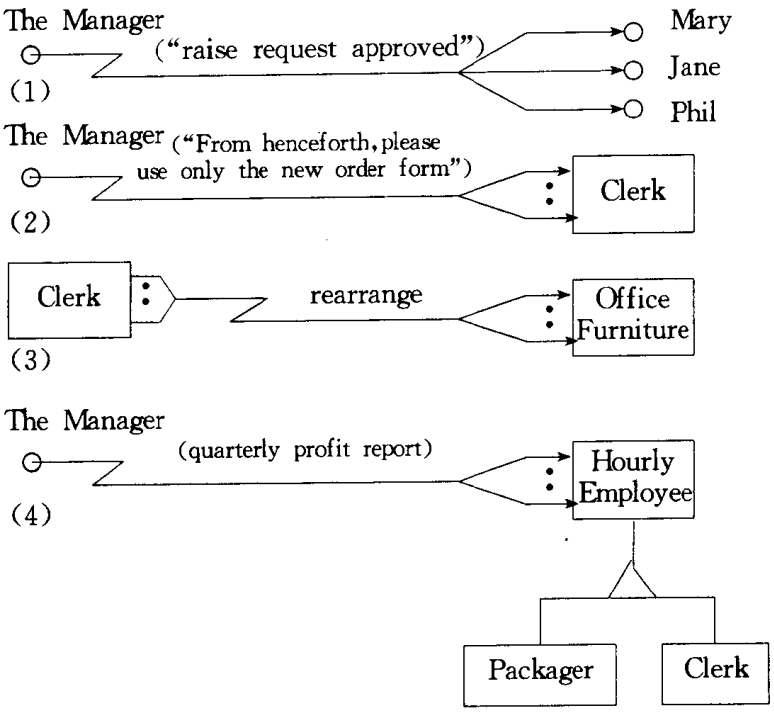


图8.35　与多个对象交互

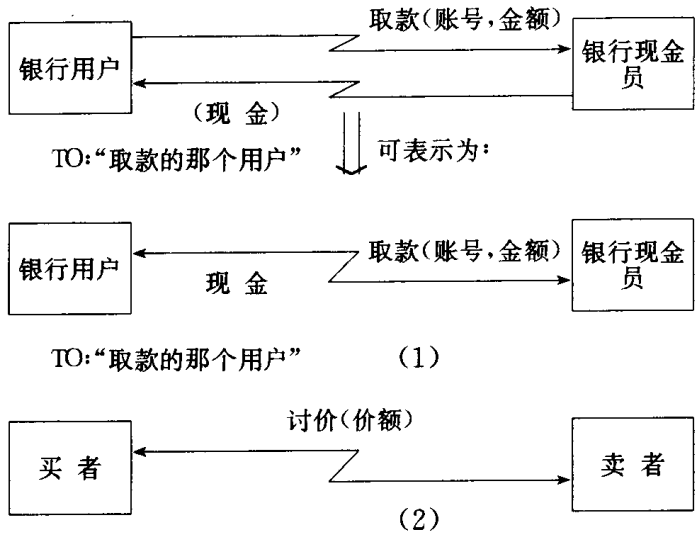


图8.36　双向交互示例

6. *特殊的交互活动*

分析员在系统分析中，常常需要"ACCESS"，"MODIFY"，"REMOVE"，"DESTROY"，"ADD"，"CREATE"这样一些活动。由于这些活动与正在建造的系统模型发生交互作用，因此把它们称为特殊的交互活动。

"ACCESS"活动用于获得对象的有关信息；"MODIFY"活动用于改变现有的对象；"REMOVE"活动用于把某一对象类中的指定对象从该类中删除，但该对象还在系统中；"DESTROY"活动用于删除一个对象；"ADD"活动用于把一个对象添加到某一对象类中；"CRE-

ATE"活动用于创建一个对象，并把该对象添加到某一对象类中。

有关这些特殊交互活动的 OSA 表示细节，可参见[Embley，1992](p178—185)。

7. "公告板"通信

通过一个公告板协议，进行对象之间的通信，这是一种常用的方法。一个对象"贴出"一条消息，其它对象"阅读"之。我们可以使用"ACCESS"，"DESTROY"，"CREATE"等，建立"公告板"通信的模型。图8.37给出了这类通信的例子。

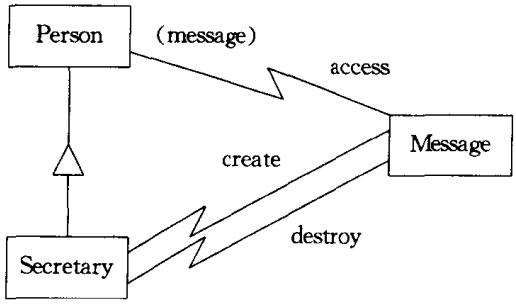


图8.37 "公告板"通信

其中，"Secretary"类的对象可以创建"Message"类的对象，也可以删除"Message"类的对象。"Person"类的一个对象可以访问"Message"类的所有对象，也可以阅读"secretary"对象当前创建的"Message"对象。

系统中，无论是概念实例，还是具体实例，有关它们之间的"公告板"通信，均可采用这一模型化技术。

8. *穿越模型边界的交互*

如果一个交互，其起始对象或终点对象不在分析模型中，即一个对象或消息从一个未被说明的起始处进入分析模型，或一个对象或消息从分析模型进入尚未说明的终点，OSA 把这样的交互称为穿越模型边界的交互。

OSA 用缺"头"或缺"尾"的交互链来表示这一类型的交互。如图8.38所示。

图8.38 超越边界的交互

9. *不间断的交互*

交互可以是连续不断的。例如，巡航控制系统中的速度传感器，它连续不断地检测航行速度，并不间断地显示当前速度。实际中，往往是尽管起始对象不断地发送消息，但终点对象并非不间断地接收之。例如，我们带的手表，它不断地"报告"时间，而我们只在需要时才"接收"手表"发来的消息"。

对于这样一类的交互，OSA 使用双箭头线表示。如图8.39所示。

10. *同类对象之间的交互*

有时，我们需要建立同类对象之间的通信模型。同类对象之间的通信可以分为两种情况。

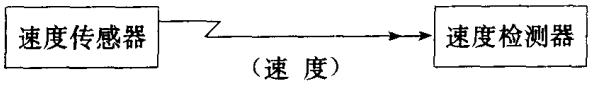


图8.39 不间断交互示例

一种是一个对象与其自身进行通信;另一种是同类对象相互之间的通信。

对于第一种情况,OSA 借助于该对象所对应的状态网,用状态或状态转换来描述一个对象与自身的通信。

对于第二种情况,OSA 使用该类状态网的不同拷贝来描述同类对象中不同对象之间的交互。除此之外,还可以使用ORM 层上的对象类,来表示同类对象中不同对象之间的通信。如图8.40所示。

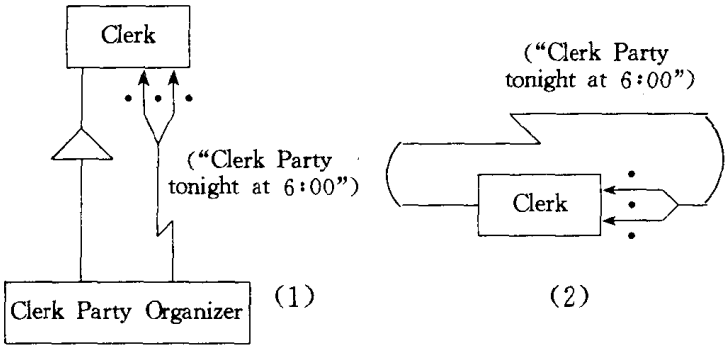


图8.40 同一对象类中的对象交互

8.3.3 交互的约束、继承

1. 交互的继承

"一般/特殊"是系统模型化的抽象方法和组织方法。它提供了继承机制。特殊类的一个对象可以继承一般类对象所具有的性质。对于交互而言,也可以应用这一机制,使所有特殊对象继承为一般对象所定义的交互,并且还具有一般对象所没有的交互。图8.41给出了这样的一个例子。

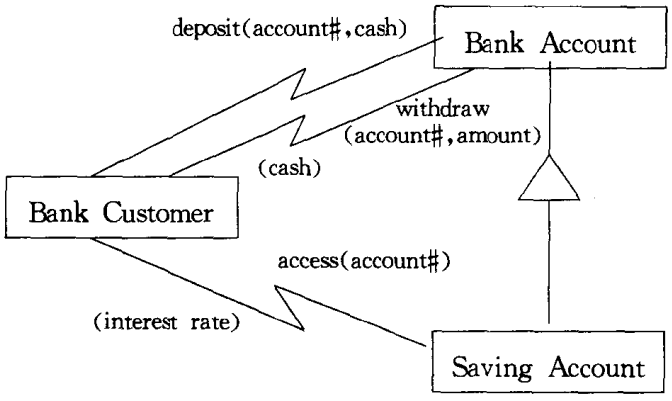


图8.41 交互的"一般/特殊"

其中,"银行账"是"存款账"的一般类。由于"存款账"是"银行账"的特殊类,因此它继承了

“银行账”的所有交互，即图中的“存款”、“取款”等。从而用户可以与“存款账”发生“存款”、“取款”交互。这些交互是隐式的，不必在交互模型中标出。另外，特殊类中的对象还可以直接与其它对象发生交互。例如图8.41中的用户可以访问“存款账”，以便了解当前的利率。由于不是所有的“银行账”均具有利率信息，因而要了解这一信息就不能与“银行账”发生交互。

2. 交互的约束

OSA 提供了两种关于交互的约束。一种是交互的实时约束，一种是交互的一般约束。

交互的实时约束限制了完成交互所需要的时间。完成交互意指参与交互的所有起始对象和终点对象均实现了它们自己的作用。在 OSA 中，交互实时约束的形式与对象行为的实时约束一样。如图8.42所示。

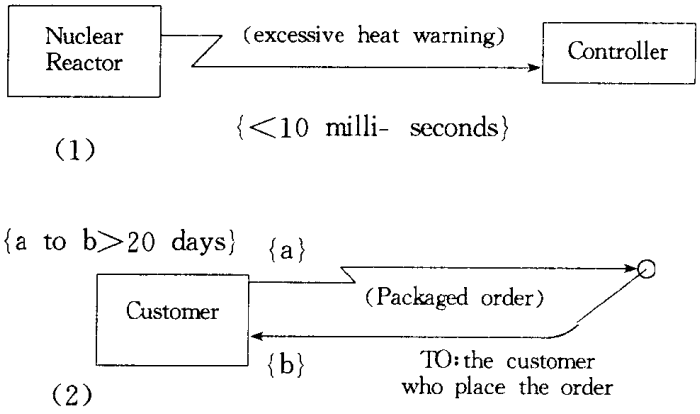


图8.42 限制时间的交互

在图8.42(2)中，{a}，{b}是交互序列的起始标志和终止标志。约束{a to b<20 days}意指在20天内必须完成这一交互序列。

交互的一般约束用于强调各种不同的交互问题。例如，在通信模型中，可以使用一般约束来限制通信频率、通信质量以及消息发送和消息接收的时间等；在异步交互模型中，可以使用一般约束来规定哪个对象具有较高的交互优先级。图8.43中给出了这一用途的一般约束。

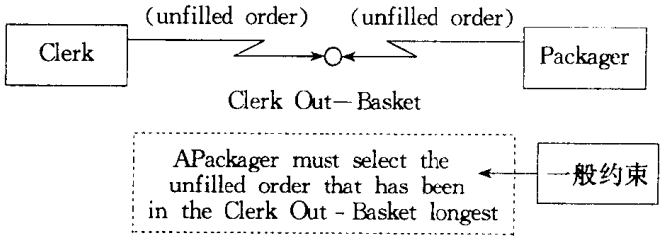


图8.43 一般约束示例

8.3.4 对象交互模型小结

在 OSA 的交互模型中，主要涉及了以下三个问题：

① 交互的基本元素以及描述对象之间交互的基本格式；

② 如何建立各种不同交互类型的模型，如同步交互、异步交互等；

③ 交互约束和继承。

OSA 提出的交互基本格式，表达了交互的本质。各种不同交互类型的表示，显示了这一基本格式的表达能力。从而，形成了 OSA 对象交互模型的独特风格。

交互模型，无论是面向过程的交互模型，还是面向对象的交互模型，都是分析模型的一个重要组成部分。

结构化分析使用了面向过程的交互模型，并且使用 DFD 表示之。

OOA[Coad，1991]的交互模型是基于消息连接的，表示了一个对象对其它对象服务的相关性。OMT[Rumbaugh，1991]使用事件跟踪，建立对象之间事件相互作用模型，并使用面向过程的 DFD 来表示对象间的信息交互。

OSA 与其它交互模型的最大区别是允许灵活地描述大量类型的交互。

图8.44及图8.45是 OSA 对象交互模型中所用的表示符号。

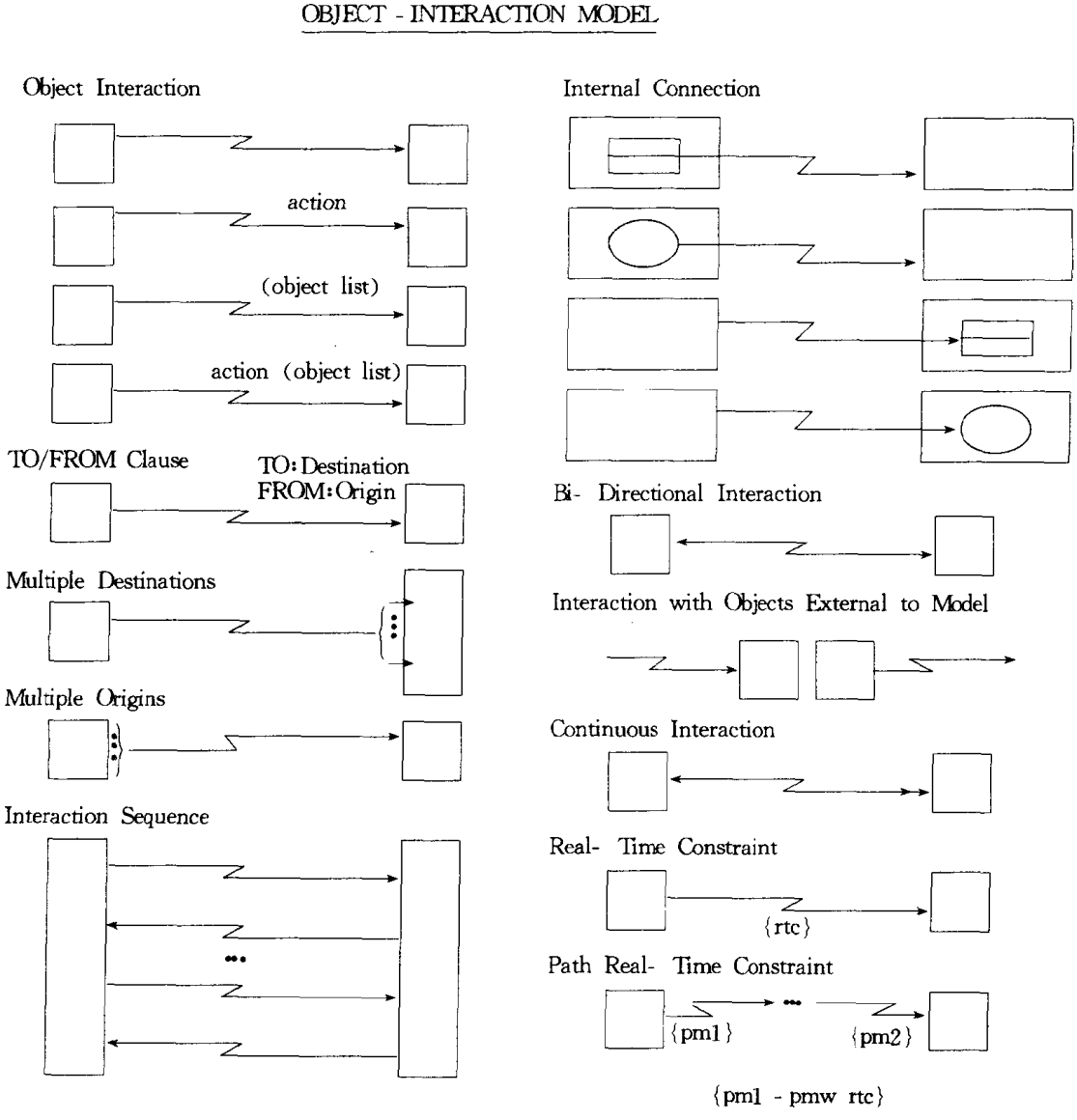


图8.44 OSA对象交互模型中所用的表示符号(1)

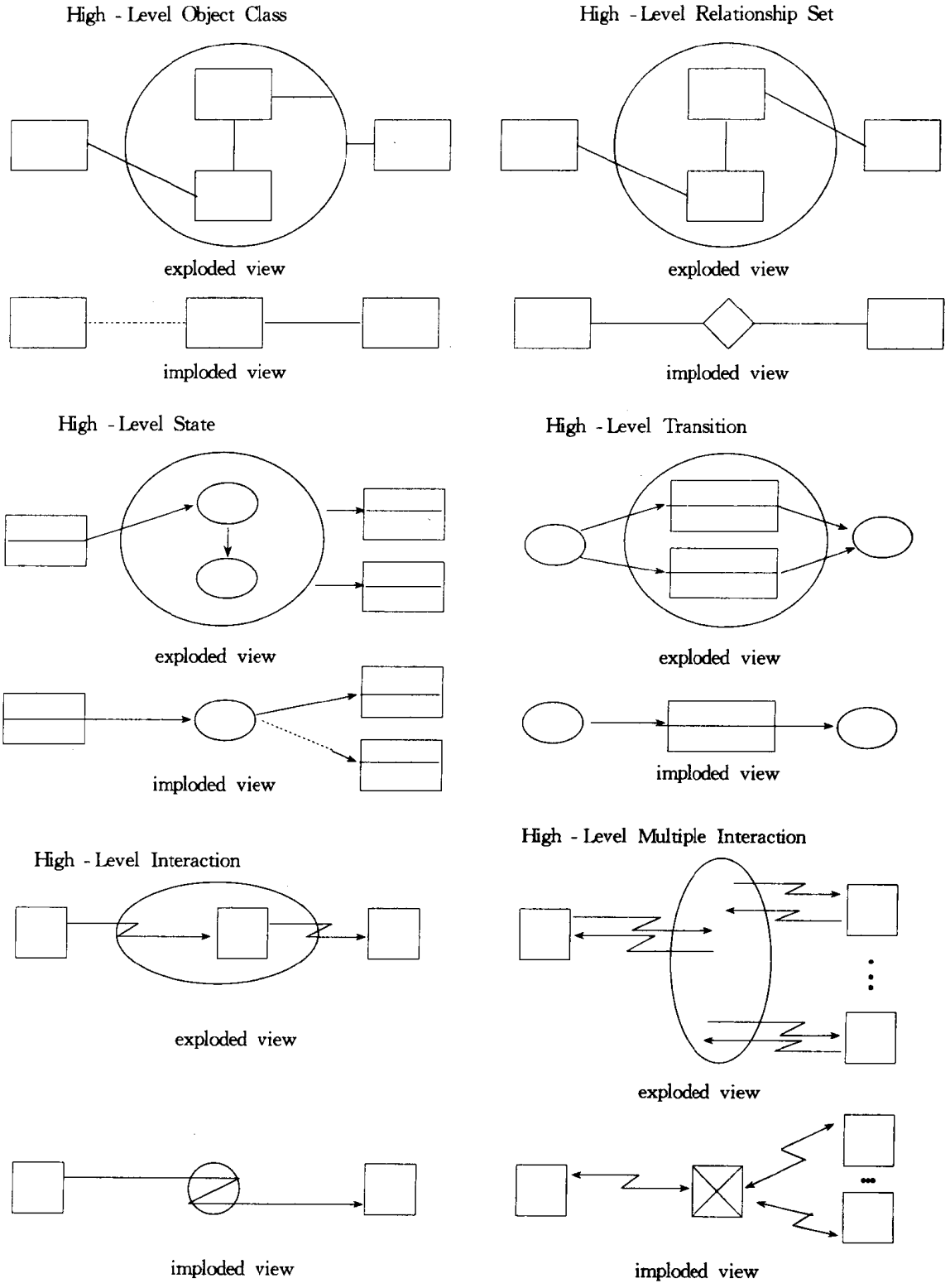


图8.45 OSA对象交互模型中所用的表示符号(2)

第九章　软 件 测 试

软件产品与其它产品不同，其最大的成本是检测软件错误、修正错误的成本，以及为了发现这些错误所进行的设计测试程序和运行测试程序的成本。据有关统计，软件测试作为软件质量保证的一个重要组成部分，在整个软件开发中占据了一半或一半以上的工作量，因此，对软件测试技术的研究一直是人们关注的课题，

本章主要针对程序测试，介绍两种常用的测试技术——基于“白盒”的路径测试技术和基于“黑盒”的事务处理流程测试技术。

9.1　软件测试目标与软件测试过程模型

9.1.1　软件测试目标

关于软件测试目标，人们在长期的实践中逐渐有了一个统一的认识。一般地说，其第一目标是预防错误。如果能够实现这一目标，那么就不需要修正错误和重新测试。可惜的是，由于软件开发至今离不开人的创造性劳动，这一目标几乎是不可实现的。因此，测试的目标即第二目标只能是发现错误。

软件错误的表现形态是多种多样的，并且，不同的错误可以有同样的表现形态，因此，即便知道一个程序有错误，也可能不知道该错误是什么。这样，要实现第二目标，也需要研究软件测试理论、技术和方法。

人们关于软件测试目的的认识，大体经历了五个阶段。第一阶段认为软件测试和软件调试没有什么区别；第二阶段认为测试是为了表明软件能正常工作；第三阶段认为测试是为了表明软件不能正常工作；第四阶段认为测试仅是为了将已察觉的错误风险减少到一个可接受的程度；第五阶段认为测试不仅仅是一种行为，而是产生低风险软件的一种认识上的训练。

根据以上讨论，软件测试可定义为：按照特定规程，发现软件错误的过程。在 IEEE 提出的软件工程标准术语中，对软件测试下的定义是：“使用人工或自动手段，运行或测定某个系统的过程，其目的是检验它是否满足规定的需求，或是清楚了解预期结果与实际结果之间的差异。”

着重指出的是，软件测试和软件调试在目的、技术和方法等方面存在着很大区别，主要表现在以下几个方面：

① 测试从一个侧面证明程序员的“失败”；而调试是为了证明程序员的正确。

② 测试以已知条件开始，使用预先定义的程序，且有预知的结果，不可预见的仅是程序是否通过测试。调试一般是以不可知的内部条件开始，除统计性调试外，结果是不可预见的。

③ 测试是有计划的，并要进行测试设计；而调试是不受时间约束的。

④ 测试是一个发现错误、改正错误、重新测试的过程；而调试是一个推理过程。

⑤ 测试的执行是有规程的，而调试的执行往往要求程序员进行必要推理以至知觉的“飞跃”。

⑥ 测试经常是由独立的测试组在不了解软件设计的条件下完成的；而调试必须由了解详

细设计的程序员完成。

⑦ 大多数测试的执行和设计可由工具支持，而调试时，程序员能利用的工具主要是调试器。

9.1.2 测试过程模型

软件测试是一有规则的过程，包括测试设计、测试执行以及测试结果比较等。这一过程见图9.1。

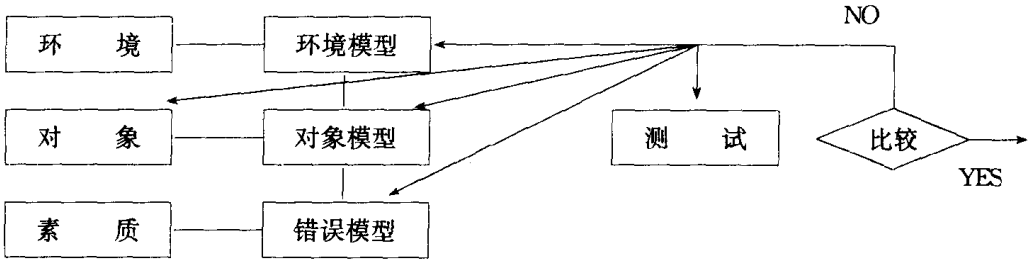


图9.1　软件测试过程模型

其中，程序环境包括支持其运行的硬件、固件和软件，例如计算机、终端设备、网卡、操作系统、编译系统、实用程序等。一般来说，程序环境经过了生产厂家的严格测试，出现错误的概率比较小，软件可靠性较好。因此，对环境的抽象——环境模型，只考虑计算机指令系统、操作系统宏指令、操作系统命令以及高级语言语句等。

另外，为了测试，我们必须简化程序概念，形成被测对象的简化版本，即程序模型。不同测试技术，对同一被测对象——程序，可产生不同的程序模型。这一简化或着重于程序的控制结构，或着重于处理过程，于是形成了所谓的“白盒”测试和“黑盒”测试。如果程序的简单模型不能解释未料到的行为，则必须修改程序模型，使其包含更多的事实和细节。如果还有问题，则要考虑是否修改程序。

由于参与软件开发的人员众多，且各有各的侧重面，因此，他们对“什么是错误”往往在认识上是不一致的。有的问题，对开发者来说，它称不上是一个“错误”，而对测试人员来说，它就是一个“错误”。为了统一认识，必须定义“什么是错误”，即给出“错误模型”。

在建立了环境模型、程序模型、以及错误模型的基础上，才能执行测试及测试结果的比较。如果预料结果与实际结果不符，就要考虑是否是环境模型、程序、程序模型和错误模型的问题。

9.2　软件测试技术

软件测试技术大体上可分为两大类：一类是白盒测试技术，典型的是路径测试技术；一类是黑盒测试技术，又称为功能测试技术，包括事务处理流程技术、状态测试技术、定义域测试技术等。白盒测试技术依据的是程序的逻辑结构，而黑盒测试技术依据的是软件行为的描述。在具体介绍这两种技术之前，首先让我们讨论一下软件错误的分类。

关于软件错误分类，至今没有统一的标准。针对本章介绍的测试技术和软件体系结构，可以把软件错误分为结构错误、数据错误、编程错误和接口错误。其中结构错误包括逻辑错误、数据流错误和初始化错误等；接口错误包括内部接口错误、外部接口错误、资源管理错误、操作系

统错误、集成化错误以及系统错误等。

人们在长期测试实践中，首先提出了路径测试技术，以发现软件中的错误。

9.2.1 路径测试技术

如上所述，路径测试技术依据的是程序的逻辑结构。表达这一结构的有力工具是控制流程图。通过合理地选择一组穿过程序的测试路径，以实现达到某种测试度量。例如，选择足够的路径，确保每一个语句至少执行一次。

路径测试对错误的假定是软件通过了与预想不同的路径。

1. 基本概念

(1) 控制流程图

控制流程图是程序控制结构的图形表示，其基本元素是过程块(简称过程)、结点、判定。图9.2即为一例。

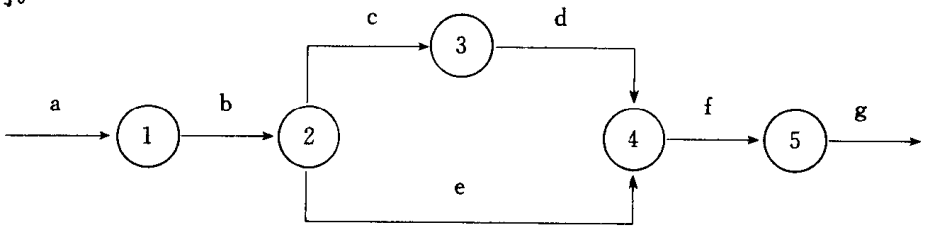


图9.2 控制流程图

其中，①，③，⑤是过程块，②是一个判定，④是一个结点。过程块是既不能由判定，也不能由结点分开的一组程序语句。其基本属性是：如果过程块中的某个语句被执行，那么块中的所有语句都被执行。按照测试的观点，对于一个过程块，如果其操作细节不影响控制流，那么这些操作就被视为是不重要的；如果其操作细节影响控制流，那么这一影响只能在其后的判定中表现出来。

判定是一个程序点，此处控制流可以分叉。在一个判定中，可以包含处理成分。判定可以是二分支的，也可以是三分支的。按照测试的观点，判定和语言上的判定语句没有本质上的差异。

结点是程序中的一个点，此处控制流可以结合。例如，汇编语言中跳转指令的目标，PASCAL 语言中的语句标号。

控制流程图与程序流程图之间的差异是在控制流程图中，不显示过程块的细节，而在程序流程图中，着重于过程属性的描述。

(2) 路径

路径是一串指令或语句。它在一个入口、结点、判定处开始，在另一入口(或同一入口)、结点、判定或出口处结束。显然一条路径可一次或多次地穿过几个结点、过程块或判定。

路径是由链支组成的。链支由其连接的“结点对”命名。例如图9.2中的(①，②)，(④，⑤)等；也可以直接命名，例如 b，c，…。路径的长度由其链支数目决定。对于软件测试，路径的严格含义是它在程序的入口处开始，在出口处结束。

2. 路径测试策略

为了回答什么是“完整的测试”，路径测试有着各种策略，基本的三种策略是：

① 路径测试(PX)：执行所有可能的穿过程序的控制流程路径。一般来说，这一测试严格

地限制为所有可能的入口/出口路径。如果遵循这一规定，则我们说达到了100%路径覆盖率。在路径测试中，该策略是最强的，但一般是不可实现的。

② 语句测试(P1)：至少执行程序中所有语句一次。如果遵循这一规定，则我们说达到了100%语句覆盖率(用C1表示)。

在路径测试中，该判据是最弱的。对于新的软件，至少要实现这一策略。

③ 分支测试(P2)：至少执行程序中每一分支一次。如果遵循这一规定，则我们说达到了100%分支覆盖率(用C2表示)。

对于结构化软件，分支测试及分支覆盖率严格地包括了语句覆盖率。

在以上三种策略中，没有涉及一个判定所依据的变量值。如果所有判定所依据的变量值以及该判定相关的过程，与其它变量没有任何关系，则可以得到该判定的各种组合，所以路径是可达的；如果所有判定所依据的变量值以及该判定相关的过程，与其它变量有一定关系，此时有的路径就可能是不可达到的。

在路径测试中，判定处求值的逻辑函数，称为判断。例如

"A>0"。

显然，程序在一个判定之后的行为取决于该判定相关变量的值。实际上，每一选取的路径都有一组布尔表达式，称为路径判断表达式。例如，

$$\begin{cases} X1+3X2+17\geqslant 0 \\ X3=17 \\ X4-X1\geqslant 14X2 \end{cases}$$

对于这一路径判断表达式，或有解，或有无穷多解，或无解。如果无解，那么该路径是不可达的。

对于路径判断表达式，如果能够找到一种有效的办法，使其路径是可达的，这一过程称为路径敏化。路径敏化一般来说，不存在一种通用的算法，在实际测试中，往往从几个不同的角度出发，力求找出不可解的问题。

显然，判断覆盖率是更强的一种覆盖。

3. 路径选取

在 IEEE 单元测试标准中，最小的强制性测试需求是语句覆盖率。在 IBM 单元测试标准中，最小的强制性测试需求是语句和分支覆盖率。

为了得到C1+C2覆盖，要选取足够的路径，图9.3即为一例。

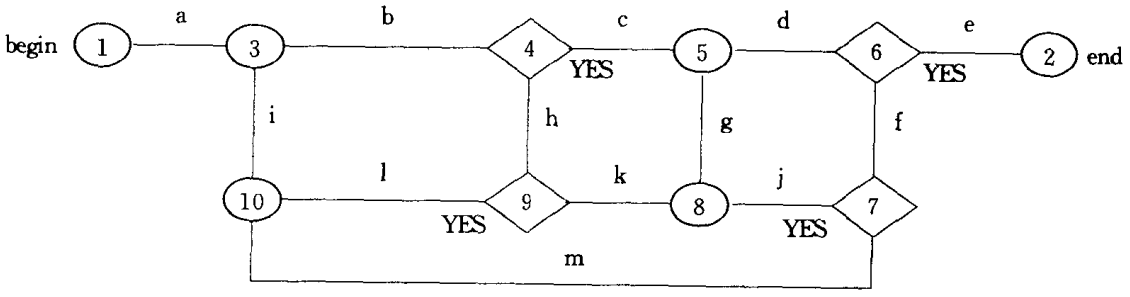


图9.3 路径选取示例

要选取如下路径：ABCDE，ABHKGDE，

ABHLIBCDE，ABCDFJGDE，

ABCDFMIBCDE

路径选取的一般规则是：

① 选择最简单的、具有一定功能含义的入口/出口路径；

② 对已选的路径进行演化，选取无循环的路径；选取短路径、简单路径；

③ 选取没有明显功能含义的路径，此时要研究这样的路径为什么存在，为什么没有通过功能上合理的路径得到覆盖。

对于循环，为了选取路径，我们把循环进行分类："单循环"、"嵌套循环"、"级联循环"和"混杂循环"，并针对不同类的循环，给出相应的路径选取规则：

(1) 单循环

单循环又可分为以下三种情况：

① 最小循环次数为零，最大次数为N，且无"跳跃"值

选取"旁通循环"(零次循环)的路径；

对循环控制变量指定为负数；

一次通过循环；

典型的重复次数；

重复次数为N－1；

重复次数为N；

重复次数为N＋1。

② 非零最小循环次数，且无"跳跃"值

重复次数为最小次数减1；

重复次数为最小次数；

重复次数为最小次数加1；

一次通过循环，除非已经覆盖；

二次通过循环，除非已经覆盖；

典型的重复次数；

重复次数为最大次数减1；

重复次数为最大次数；

重复次数为最大次数加1。

③ 具有跳跃值的单循环

除把每一"跳跃"边界，按"最小循环次数"和"最大循环次数"处理外，其它均同前两种单循环的路径选取规则。

(2) 嵌套循环

嵌套循环的路径选取策略是：

① 在最深层的循环开始，设定所有外层循环取它的最小值；

② 测试最小值、最小值＋1、典型值、最大值－1、最大值，与此同时，测试最小值减1、最大值＋1以及"跳跃值"边界；

③ 设定内循环在典型值处，按②测试外层循环，直到覆盖所有循环。

（3）级联循环

如果在退出某个循环以后，到达另一个循环，且还在同一入口/出口路径上，则称这两个循环是级联的。其中，如果在退出某个循环以后，到达另一个循环，且循环的重复值与另一个循环的重复值相关，该循环还在同一入口/出口路径上，则可认为是嵌套循环。如果两个循环不在同一入口/出口路径上，则可认为它们是单循环。

综上所述，路径测试是一种强有力的单元测试技术。路径测试的目标是：执行足够的测试，以确保最小的C1+C2覆盖率。在路径测试中，一般要从最简单、最熟悉的从入口到出口的路径以及与正常路径有偏差的路径开始选取，继之按需要增加路径，以达到一定的覆盖率。并且对于循环还要适当地增加一些路径，以覆盖循环和循环组合：

无循环

一次循环

二次循环

比最大值小1

最大值

最大值加1

最小值减1

9.2.2 事务处理流程测试技术

事务处理流程是系统行为的一种表示方法，为功能测试建立了程序的动作模式。其中，使用了控制流程的概念成分，例如链支，结点等。应该说，不论是结构测试，还是功能测试，基于结点、链支的图形表示技术都是一种强有力的概念工具。这种测试技术的基本步骤可概括为：定义有用的图形模式，设计必要的测试用例以覆盖之。

1. 事务与事务流程

事务是从系统用户的角度出发所见到的一个工作单元。一个事务由一系列操作组成，其中，某些操作由系统执行，某些操作由用户或系统之外的设备执行。例如，在联机信息检索系统中，一个事务可以是：

接受输入

效输入

向用户发送礼节性信息

进行输入处理

检索文件

向用户请求指令

接受输入

有效输入

处理请求

更改文件

传送输出

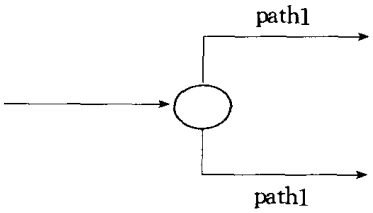
记录事务注册和清除(结束)

由上可见，事务有其“产生”和“消亡”。

一个系统的行为表现为多个事务的执行，这一行为可抽象为事务处理流程图。

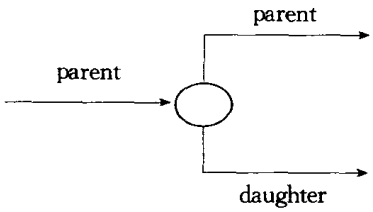
事务处理流程图与控制流程图的类同点是使用了相同的概念成分，例如处理（对应于过程块）、分支、结点。它们不同之处是，事务流程图是一种数据流程图，从操作应用的历史，观察数据对象。因此，链支和过程块的定义有所差异。另外事务流程图的判定结点可能是一个复杂的过程，从而事务流程图中的判定只能是“抽象”。第三点不同之处是事务流程图中存在“中断”的作用，中断可以把一个过程等价地变换为具有繁多出口的链支，对此也要予以抽象。具体地说：

事务流程图中的判定可以是：



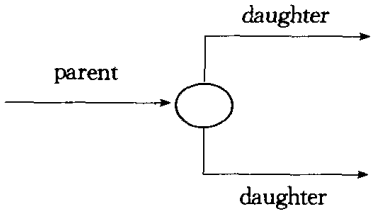
即事务处理将选取一个分支执行，其语义与控制流程图的分支相同。

事务流程图中的判定也可以是：



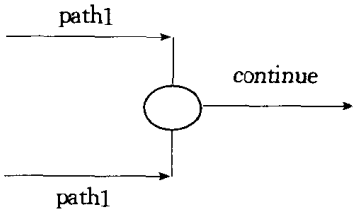
即事务处理产生一个新事务，由此这两个事务继续执行，称为“并生”。

事务流程图中的判定还可以是：



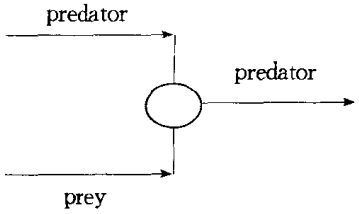
即事务处理产生两个新事务，称为“丝分裂”。

事务流程图的结点可以是：



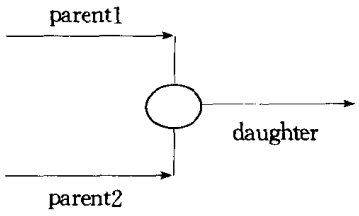
即事务的不同活动可以汇集一处，其语义与控制流程图相同。

事务流程图的结点也可以是：



即一个事务可以被另一事务“吸食”，称为“吸收”。

事务流程图的结点还可以是：



即两个事务结合后生成一个新的事务，称为“结合”。

由此可见，事务流程图要比控制流程图在语义上更为复杂。

事务处理流程图往往具有很差的结构，其主要原因是：

① 它是一种处理流程，人可包含在循环、判定中；

② 某些部分可能与我们不能控制的行为有关；

③ 性能的增加，可使事务数目和单个事务处理流程具有相当的复杂性；

④ 事务流程表达的系统模型更接近现实，例如中断、多任务、同步、并行处理……，所有这一切已不再适合结构化概念。

2. 事务处理流程测试步骤

事务处理流程测试的步骤大体分为以下三步：

第一步：获得事务处理流程图

由于在实际的软件工程中，不论是系统分析规约，还是系统设计规约，均采用了非形式化的表达技术，因此要获得测试可用的事务处理流程通常是最困难的。

第二步：浏览、复审

这一步主要是对事务分类，为测试用例的设计奠定基础。

第三步：用例设计

设计足够的测试用例，实现C1+C2覆盖。并且

① 循环、最大值、域界，选取附加的事务处理路径；

② 具有很大危险或潜在危险的事务选取附加的路径，包括“并生”、“丝分裂”、“吸收”和“结合”；

③ 选择测试路径的一个子集，作为系统功能测试的基础。

对于事务处理流程测试，为了实施以上活动，要解决以下几个问题：

(1) 路径选取

选取一个基于功能的、切合实际的事务路径覆盖集，其中，找出从事务处理流程入口到出口的最复杂、最长和异常的路径。特别是对那些异常路径，以期发现：

遗漏互锁

重复互锁

接口

交叉影响

重复处理等问题

并建立路径目录。

(2) 激活

一般来说,80%—95%的路径(C1+C2)容易激活,但余下的路径是不易激活的。不易激活的路径一般来说或是事务流程的错误,或是设计上的错误。

(3) 测试设备

为了支持事务处理流程测试,应配备中心事务调度设备和事务跟踪装置,以便有效地实施测试。

(4) 测试数据库

由于事务处理流程测试的复杂性,应建立相应的测试数据库,以保留测试数据,支持有效的测试配置。

综上所述,事务处理流程测试技术是将路径测试技术用于功能测试的结果,有关单元和程序的路径测试方法可用于解释基于事务处理流程的功能测试;对于事务处理流程测试而言,最大的问题和最大的代价是获得事务处理流程图;一般地,事务处理流程测试要求达到C1+C2路径覆盖;但是,大多数错误将在奇异的、不受注意的或非法的路径上发现;更为重要的是,在事务处理流程测试中,如果设计的测试能与设计者讨论,将可以发现比运行测试更多的错误。

9.3 软件测试步骤

由于软件错误的复杂性,在软件工程测试中我们应综合运用测试技术,并且应实施合理的测试序列:单元测试、集成测试、有效性测试和系统测试。单元测试集中于每个独立的模块;集成测试集中于模块的组装;根据软件有效性的一般定义:软件实现了用户期望的功能,有效性测试集中检验是否符合用户所见的文档,包括软件需求规格说明书,软件设计规格说明书以及用户手册等;系统测试集中检验系统所有元素(包括硬件、信息等)之间协作是否合适,整个系统的性能、功能是否达到。其中,系统测试以超出软件测试,属于计算机系统工程范畴。

下面简单介绍一下与软件测试有关的单元测试、集成测试以及有效性测试。

9.3.1 单元测试

单元测试主要检验软件设计的最小单位——模块。该测试以详细设计文档为指导,测试模块内的重要控制路径。一般来说,单元测试往往采用白盒测试技术。

在单元测试期间,通常考虑模块的以下四个特征:

① 模块接口;

② 局部数据结构;

③ "重要的"执行路径;

④ 错误执行路径;

以及与以上四个特性相关的边界条件。

单元测试首先测试穿过模块接口的数据流，为此应当测试：输入实际参数的数目是否等于形式参数的数目、实际参数的属性与形式参数的属性是否匹配、实际参数的单位与形式参数的单位是否一致、传送给被调用模块的形式参数数目是否等于实际参数的数目、传送给被调用模块的形式参数属性是否与实际参数属性匹配，传送给被调用模块的形式参数单位是否与实际参数单位一致、对实际参数的任何访问是否与当前的入口无关、跨模块的全程变量定义是否相容等。如果该模块是实现外部I/O的模块，还必须测试：文件属性是否正确；I/O语句与格式说明是否匹配；记录长度与缓冲区大小是否匹配，是否处理了文件结束条件；是否处理了I/O错误等。

继之，进行数据结构的测试。为此，要设计相应的测试用例，以发现下列类型的错误：不正确的或不相容的说明、置初值的错误或错误的缺省值、错误的变量名、不相容的数据类型、下溢与上溢错误等。除了局部数据结构外，还应确定全程数据对模块的影响。

第三，还要进行执行路径的选择测试。为此，也要设计相应的测试用例，以发现由于不正确的计算、错误的判定或错误的控制流而引发的错误。常见的错误有：算术运算优先级错误、置初值错误、表达式符号表示错误、计算精度错误、不同的数据类型进行比较、循环终止错误（包括循环出口错误）、不正确地修改循环变量等。

边界测试是单元测试中的最后工作，往往也是最重要的工作，因为软件常常在边界上出现错误。

在单元测试中，由于模块不是一个独立的程序，必须为每个模块单元测试开发驱动模块和（或）承接模块。驱动模块模拟“主程序”，接受测试用例的数据，将这些数据传送给要测试的模块，并打印有关的结果。承接模块代替被测模块的下属模块，打印入口检查信息，并将控制返回到它的上级模块。

驱动模块和承接模块作为单元测试的测试设备，需要花费一定的开销进行编制。

当被测模块的设计是高内聚的或是一个功能性模块时，单元测试就比较简单，只要设计少量的测试用例，便会容易地预计结果和发现错误。

9.3.2 集成测试

每个模块完成了单元测试，把它们组装在一起并不一定能够正确地工作，其原因是模块的组装存在一个接口问题。具体表现在：数据通过接口时可能予以丢失；一个模块可能对另一模块产生“副作用”；子模块的组合可能不实现所要求的主功能；模块与模块之间的误差积累可能产生不可接受的程度等等。

集成测试是软件组装的一个系统化技术，其目标是发现与接口有关的错误，将经过单元测试的模块构成一个满足设计要求的软件结构。

集成测试可“自顶向下”地进行，称为自顶向下的集成测试；也可以“自底向上”地进行，称为自底向上的集成测试。

自顶向下的集成测试是一种递增地组装软件的方法。从主控模块（主程序）开始，沿控制层次向下，或先深度或先宽度地将模块逐一组合起来，形成与设计相符的软件结构。

对于先深度的集成测试，依据应用的专业特性，选取结构的一条主线（路径），将相关的所有模块组合起来。见图9.4所示例子。

可以选择左边的路径作为主线，首先组合模块M1，M2，M5和M8，然后组合M6（如果

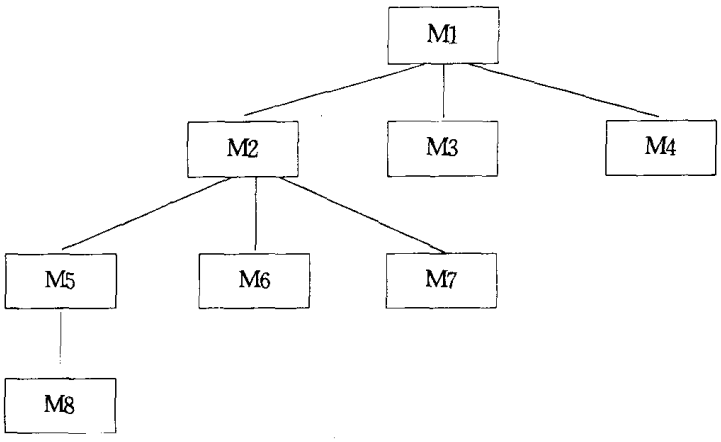


图9.4　自底向上的集成测试

M2的某个功能需要)M6，继之再构造中间和右边的控制路径。

对于先宽度的集成测试，逐层组合直接的下属模块。例如，首先组合模块M2，M3和M4，继之组合M5，M6和M7，……。

一般来说，集成测试是以主控模块作为测试驱动模块，设计承接模块替代其直接的下属模块，依据所选取的测试方式(先深度或先宽度)，在组合模块时进行测试。每当组合一个模块时，要进行回归测试，即对以前的组合进行测试，以保证不引入新的错误。

自底向上的集成测试从软件结构最低的一层开始，逐层向上地组合模块并测试。由于在给定层次上所需要的下属模块其处理功能是可以使用的，因此无需设计承接模块。

一般来说，自底向上的集成测试首先将低层模块分类为实现某种特定功能的模块组；继之，书写一个驱动模块，用以协调测试用例的输入和输出，测试每一模块组；沿着软件结构向上，逐一去掉驱动模块，将模块组合起来。这一过程如图9.5所示。

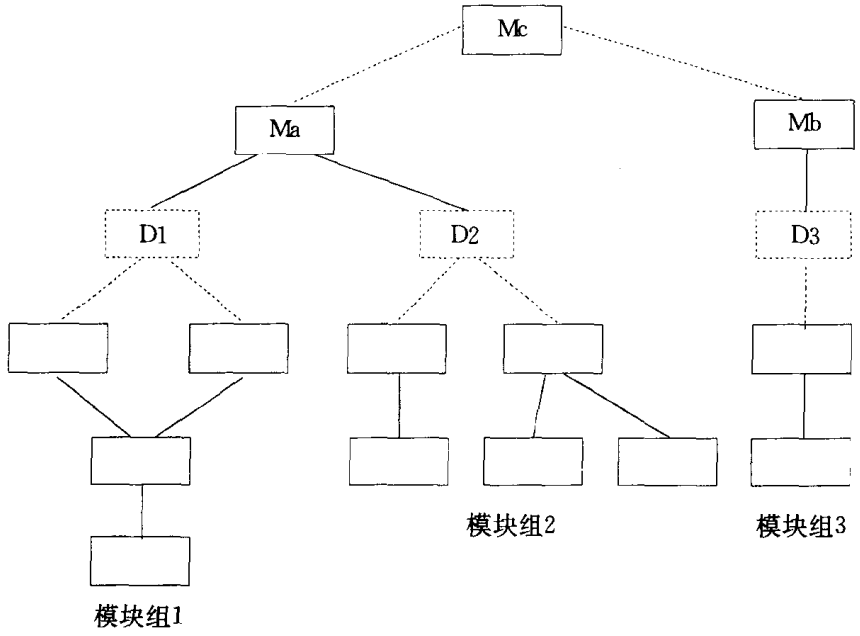


图9.5　自底向上的集成测试

其中，有三个模块组，为每一模块组设计一个驱动模块（虚线框），并进行测试；去掉驱动模块D1和D2，将这两个模块组直接与模块Ma接口，类似地，去掉D3，将另一模块组直接与模块Mb接口。

自顶向下和自底向上的集成测试各有其优缺点。自顶向下的主要缺点是需要设计承接模块以及随之而来的测试困难。自底向上的主要缺点是“只有在加上最后一个模块时，程序才作为一个实体而存在”。在实际的集成测试中，应根据被测软件的特性以及工程进度，选取集成测试方法。一般来说，综合地运用这两种方法，即在软件的较高层使用自顶向下的方法，而在低层使用自底向上的方法，可能是一种最好的选择方案。

9.3.3 有效性测试

有效性测试的目标是发现软件实现的功能与需求规格说明书不一致的错误。因此，有效性测试通常采用黑盒测试技术。为了实现有效性测试，应制定测试计划，该计划根据采用的测试技术，给出要进行的一组测试。继之进行测试用例和预期结果的设计。通常，在测试执行之前，应进行配置复审，其目的是保证软件配置的所有元素已被正确地开发并编排目录，具有必要的细节以支持软件生存周期中的维护阶段（如图9.6所示）。

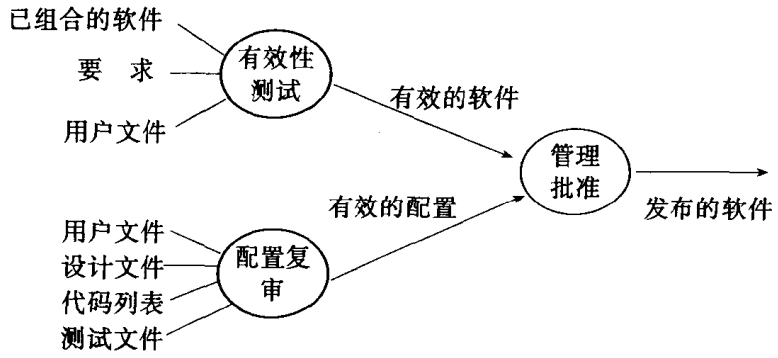


图9.6 配置复审

在测试计划完成之前，有效性测试发现的偏差或错误一般不予纠正，通常需要和开发者一起协商，以建立解决这些缺陷的方法。

9.3.4 软件测试与程序正确性证明

通过以上内容的介绍，我们可以看到，测试可以发现错误。但是，测试并不能表明程序的正确性。如果能够开发出可靠的程序正确性证明，那么测试的工作将大幅度的减少，软件危机的一个重要方面——软件质量不可靠问题，将不复存在。

程序正确性证明涉及许多复杂的领域，利用数学归纳法或谓词演算，进行人工的正确性证明，对于小程序可能有些价值（参见附录），但对于大一点的软件系统，这种证明几乎没有什么用处。正如Aderson所说：“我们清楚地认识到，人工的正确性证明可能很容易包含错误，它不是避免或发现程序所有错误的灵丹妙药。”

近20年来，已经开发了一些自动的计算机软件正确性证明方法。利用人工智能和谓词演算为基础的自动化证明技术，针对某些特定的语言（例如PASCAL，LISP等）已开发了正确性证

明程序，但是把它们实际运用于大型软件，还要进行大量的工作，需要一个相当长的时间。

综上所述，软件测试在软件开发的整个技术工作中占据了很大的比重。随着信息化需求，人们越来越认识到软件测试的含义。

软件测试的目标是揭示错误。为实现这一目标，要实施一系列的测试，包括单元测试、集成测试、有效性测试和系统测试。单元测试集中于单个模块的功能和结构检验，集成测试集中于模块组合的功能和软件结构检验。有效性测试论证软件需求的可追朔性，系统测试验证将软件融于更大系统中整个系统的有效性。

每种测试将涉及一系列系统的测试技术，以支持测试用例设计和测试执行。目前，在软件开发中，通常采用人工测试技术，但随着自动化技术的研究，日益增多的自动化工具将显示出广阔的应用前景。

9.4 程序证明技术

专业人员所编制的程序，一般来说每千行均包含10个以内独立可改正的错误。因此，Hoare 在第十八届国际软件工程会议(1996)上指出，“不借助程序证明，软件可靠性何以保证。”20多年来，人们一直重视程序正确性证明技术的研究，并取得了一定的成果。

分析近年来软件工程的实践，我们可以发现，为了提高软件可靠性，采用了与其它现代工程学大致相同的技术，包括：

- 对设计工作进行有计划的、严格的复审；
- 不断提高机器与环境性能，支持软件设计自动化；
- 为保证软件质量，进行大规模、有目标的测试；
- 适应已广泛使用的软件产品，并进行演化；

……

应当承认，现代软件工程的进步，归功于在理论、概念和方法等诸多方面的研究成果。其中，形式化技术和证明技术，也如同其它现代工程学科一样，担当了重要的角色，起到了十分重要的作用。

今天，关于程序正确性问题，反映得就不像20年前人们想像的那么严重。特别是 Mackenzie 最近研究成果表明，在几千种属于计算机系统可靠性的致命危害中，仅有10个左右的问题可归结为软件中的错误，而大多数错误源于不正确的计算方法。

如同一些安全性要求很强的工程领域，软件领域的技术成果转化为生产力的速度是很慢的，至今程序证明技术还没有很好地应用于软件生产的实践，可望在不久的将来，会呈现出一种新的状况。为此，本节简单介绍一下程序证明技术——程序的公理化证明技术。

程序的一个十分重要的性质是，它是否实现了预期的功能。为了验证这一性质，可以为程序执行之后变量所能取得的值构造一个断言，以此来表示程序或部分程序是否实现了它的预期功能。通常，这一断言要指出其中每一变量值的一般性质以及值与值之间的关系。断言可以用数理逻辑的符号来表示，其中可以使用易读且熟悉的操作优先规则。

在很多情况下，程序结果的正确性依赖于该程序启动之前变量所取的值。这些前提也可以用断言来表达。

为了指出前提(P)、程序(Q)和程序执行结果(R)这三者之间的联系，我们引入一种新的

符号。

$$P\{Q\}R$$

这个符号可理解为:“如果程序 Q 在启动前其断言 P 为真,那末在 Q 终止时其断言 R 要为真。”如果没有什么强加的前提,那末就可以把上式写为:

$$\text{true}\{Q\}R$$

1. Hoare 系统

(1) 赋值公理

赋值是计算机的一个最明显的特性。考虑如下赋值语句:

$$x := f$$

其中,x 是一个简单变量的标识符; f 是一个没有副作用的程序设计语言的表达式,但 f 中可能含有 x。

显然,赋值之后对 x 的值为真的任一断言 $P\{x\}$,也一定使赋值之前所采用的表达式 f (其中,f 中的 x 具有原来的值)的值为真。这样,如果在赋值之后 $P\{x\}$ 为真,那末在赋值之前 $P\{f\}$ 一定为真。这个事实可以更形式地表达为:

D_0赋值公理

$$\vdash \{x := f\} P$$

其中,x 是一个变量标识符; f 是一个表达式; 是通过以 f 替代 P 中所有出现的 x 而得到的。

D_0实际上不是一个公理,而是一个公理模式。该模式描述了一个无限的公理集合,其中的公理具有同一的样式。在程序证明时,我们可以把它看作是一个公理。

(2) 推论规则

除了公理之外,演绎科学至少还需要一条推导规则,该规则容许我们从一个或多个公理或已被证明的定理中演绎新的定理。一个推导规则采用形式:“**if** x **and** y **then** z”,即如果形式 x 和 y 的断言已作为定理予以证明了,那么 z 也被证明,z 即可作为一个定理。一个推导规则的最简单例子指出:如果程序 Q 的执行保证了断言 R 的真值,那么它也保证了被 R 逻辑蕴涵的每一个断言的真值;另外还有,如果对于产生结果 R 的程序 Q 而言,已知 P 是其前提,那么逻辑蕴涵 P 的另一其它断言也必是 Q 的前提。以上两个推导规则可以更形式地表示为:

D_1推论规则

if $\vdash P\{Q\}R$ **and** $\vdash R \supset S$ **then** $\vdash P\{Q\}S$

if $\vdash P\{Q\}R$ **and** $\vdash S \supset P$ **then** $\vdash S\{Q\}R$

(3) 结构规则

一个程序一般是由一组顺序的语句组成的,这些语句一个跟着一个地执行,并以分号“;”或其它等价的符号把这些语句分开:

$$(Q_1; Q_2; \cdots\cdots; Q_n)$$

为了避免冗长,可以仅讨论两个语句(;),因为多个语句可构造为如下形式:

$$(Q_1;(Q_2;(\cdots(Q_{n-1},Q_n)\cdots)))$$

与结构相联系的推导规则指出:如果程序的第一部分的证明结果与程序第二部分的前提是相等的,并在该前提下,程序产生了它的预期结果,那么,只要程序的第一部分的前提被满足,则整个程序将产生其预期的结果。该规则可形式地表示为:

D_2结构规则

if $\vdash P\{D_1\}$ **and** $\{Q_2\}R$ **then** $\vdash P\{Q_1; Q_2\}R$

(4) 迭代规则

计算机的一个主要特性是它能够重复地执行程序 S 的某一段，直到条件 B 为假。表达这一迭代的简单方法是采用 ALGOL 60 中的 **while** 符号：

while B **do** S

根据上述语句的语义，迭代推导规则公式化的推理如下：假定 p 是一个断言，只要它在 S 初始时为真，它在 S 完成时总是真的。显然在 S 语句的任一次迭代之后(甚至 S 不重复执行)p 仍然一直为真。另外，在迭代最终结束时，条件 B 为假这是已知的。依据 B 在 S 初始时可以假定为真这一事实，迭代规则可以形式地表示为：

D_3迭代规则

if $\vdash p \wedge B\{S\}p$ **then** $\vdash p\{$**while** B **do** S$\}\ \neg B \wedge p$

以上的公理和规则作了以下的限制：

① 首先，我们假定了所引用的推导规则和公理没有求值表达式和条件的副作用。在以一种容许副作用的语言所表达的程序的性质证明中，在应用合理的证明技术之前，必须证明在每一种情况下均没有副作用。

② 其次，在以上引用的公理和规则中，都没有提供证明程序成功终止的基础。就程序的终止失败而言，其原因可能是由于死循环或由于违背了实现时所固定的限制。例如数的区域、内存的大小或操作系统的时间限制等。如此，符号“$P\{Q\}R$”应解释为：“只要程序成功地终止，那么 R 描述了程序结果的性质。”修改公理以使它们不能用于预言非终止程序的结果，这是相当容易的。但是这些公理的使用现在要依赖对许多实现相关的性质的了解，例如，计算机的大小和速度、数的区域以及溢出技术的选择。

③ 最后，还有一些不予讨论的范围。例如：实数、位及字符、复数、分数、数组、记录、重定义、文件、输入/输出、说明、子程序、参数、递归以及并行执行。甚至整数算术的特性也远不是完整的。只要程序语言是简单的，那么对这些范围的处理好像没有什么太大的困难。真正困难的问题是标号以及转移、指针、名字参数。对于使用这些性质的程序，其证明要精心编制；并且暗含着构成公理基础的复杂性。

综上所述，关于顺序程序性质证明的 Hoare 系统可归纳如下：

令 P 和 Q 是关于变量的断言，并且 S 是一个语句。非形式地，符号

$$\{P\}S\{Q\}$$

的意义为：如果 P 在 S 执行之前为真，那么 Q 在 S 执行之后为真。并没有提及 S 的终止问题。只要 S 终止，则 Q 成立。

符号

$$\frac{a}{b}$$

的意义为：如果 a 为真，则 b 也为真。为了证明顺序程序的性质，Hoare 就是使用这样的符号描述了一个推演系统。

令 P, P_i 表示断言，x 表示变量，E 表示表达式，B 表示布尔表达式；并令 S, S_i 表示语句。那么对于五类所容许的语句，其公理是：

① $\{P\}$ skip $\{P\}$

空

② $\{P_E^x\}\ x := E\{P\}$ 赋值

其中 P_E^x 表示以 E 代替 P 中每一自由出现的 x 之后所产生的断言。

③ $\dfrac{\{P \wedge B\}S_1\{Q\},\{P \wedge \neg B\}S_2\{Q\}}{\{P\}\text{if S then } S_1 \text{ else } S_2\{Q\}}$ 选择

④ $\dfrac{\{P \wedge B\}S\{P\}}{\{P\}\text{while B do S}\{P \wedge \neg B\}}$ 循环

⑤ $\dfrac{\{P_1\}S_1\{P_2\},\{P_2\}S_2\{P_3\},\cdots,\{P_n\}S_n\{P_{n+1}\}}{\{P_1\}\text{ begin } S_1;\ S_2;\ \cdots;\ S_n \text{ end } \{P_{n+1}\}}$ 复合

除此之外，我们还有如下的推理规则：

⑥ $\dfrac{\{P_1\}S\{Q_1\},P \vdash P_1 Q_1 \vdash Q}{\{P_1\}S\{Q\}}$ 推理

其中符号 $P \vdash Q$ 意味着可以使用 P 作为一个假设来证明 Q。但没有给出根据 P 来证明 Q 的推演系统，它可以是对于程序设计语言中所使用的数据类型以及操作均有效的系统。

现在，让我们简单地讨论一下顺序程序性质的证明。当我们写出 $\{P\}S\{Q\}$ 时，则意味着通过使用(1)—(6)，存在着 $\{P\}S\{Q\}$ 的一个证明。例如，假设语句 S

begin x := a ; **if** e **then** S1 **else** S2 **end**

并假设我们已经有了证明

$$\{P_1 \wedge e\}S_1\{Q_1\} \text{ 和 } \{P_1 \wedge \neg e\}S_2\{Q_1\}$$

那么，$\{P\}S\{Q\}$ 的一个证明可以是：

① $\{P_{1a}^x\}\ x := a\{P_1\}$ 赋值

② $\dfrac{\{P_{2a}^x\}x := a\{P_1\},P \vdash P_{1a}^x}{\{P\}x := a\{P_1\}}$ 推理规则

③ $\dfrac{\{P_1 \wedge e\}S_1\{Q_1\},\{P_1 \wedge \neg e\}S_2\{Q_1\}}{\{P_1\}\text{if e then } S_1 \text{ else } S_2\{Q_1\}}$ 选择

④ $\dfrac{\{P_1\}\text{ if e then } S_1 \text{ else } S_2\{Q_1\},\ Q_1 \vdash Q}{\{P_1\}\text{ if e then } S_1 \text{ else } S_2\{Q\}}$ 推理规则

⑤ $\dfrac{\{P\}x := a\{P_1\},\{P_1\}\text{ if e then } S_1 \text{ else } S_2\{Q\}}{\{P\}\text{ begin x := a; if e then } S_1 \text{ else } S_2 \text{ end } \{Q\}}$ 复合

下面给出 这个证明的另一种证明形式。在这一证明形式中，由于程序伴随着适当位置上的断言给出，因此其证明就容易理解得多。在这样的证明形式中，两个相邻的断言 $\{P_1\}\{P_2\}$ 表示推理规则的一种用法，在推理中，它们被写为 $P_1 \vdash P_2$。

```
⊢ {P}
begin {P}
      {Pl_a^x}
      x := a;
      {P1}
      if e then{ P1 ∧ e}
                S1
                {Q1}
      else {P1 ∧ ¬ e}
           S2
```

{Q1}

{Q1}

{Q}

end

{Q}

以上给出的证明形式及程序执行时性质的证明依赖于以下性质：

令 S 是程序 T 中的一个语句，并且 pre(S)是证明形式 {P}T{Q}中 S 的前置条件。假设 T 的执行以 P 为真开始并达到即将执行 S 的那一点，此时，所有变量处于状态 m 之中，那么 pre(S)[m] = true。

2. *并行程序的正确性证明*

为了引入并行性，我们对顺序语言予以扩充，使之具有两个新的语句。一个是为并行处理而引入的 cobegin 语句，另一个是为了协调并行执行的进程而引入的 await 语句。

令 $S_1, S_2, \cdots, S_n$ 是一些语句，则 cobegin 语句：

cobegin $S_1 \| S_2 \| \cdots \| S_n$ **coend**

的执行导致了语句 S_i 的并行执行。当所有进程 S_i 的执行终止时，则 cobegin 语句的执行才终止。对于并行执行的实现方法没有什么限制；尤其是，对于这些进程之间的相对速度更没有任何假定。

我们要求每个赋值语句的执行和每一表达式的求值要作为一个独立的、不可分割的活动。但如果程序附有如下简单的约定(本节讲述的内容遵循这一约定)：

(1) 每个表达式 E 求值时，它至多可以引用一个可以被另一进程改变的变量 y，且至多引用一次。对于赋值语句 x :=E，具有类似的限制。

依据(1)的约定，所需要的唯一不可分活动是“内存引用”，即假定进程 S_i 引用变量(地址) c 而另一进程 S_j 正改变 c。我们要求 S_i 接受的 c 值，或是对 c 赋值之前的值，或是对 c 赋值之后的值，但 c 值不能由于赋值期间 c 值的“波动”而导致不真。因此，我们的并行语言就可以用于模拟在任一适当的机器上的并行执行。

我们引入的第二个语句是：

await B **then** S

其中 B 是一个布尔表达式，S 是一个不含有 cobegin 语句或含一 await 语句的语句。当一个进程企图执行 await 时，则该进程被延迟到 B 为真。因此语句 S 是作为一个不可分的活动执行的。当 S 终止时，并行处理继续。如果两个或两个以上进程正等待同一条件 B，那么当 B 变为真时，只允许它们中的一个进程优先处理，而其它进程则继续等待。在一些应用中，要求给出调度等待进程的次序；但就我们的目的而言，其调度进程的规则是任意的。注意，B 的求值是 await 语句这一不可分割活动的一组成部分；另一进程在 B 求值之后，在 S 开始执行之前均不可改变变量以使 B 成为假。

使用 await 语句可使任一语句 S 成为一个不可分割的活动：

await true **then** S

或把 await 语句纯粹用作一个同步工具：

await “某个条件” **then** skip

注意，我们宁愿把 await 语句作为表达多个标准的同步原语(例如信号量)的工具提出，而不把

它作为插入到其它程序设计语言中的新的同步语句，因为要求太强，而不能有效实现。这样，为了验证使用信号量的程序，首先以 await 表达信号量操作，随之应用本节给出的技术。

现在，我们再去形式地定义(2)，(3)中的语句。await 的定义是一目了然的，但关于 cobegin 语句 D 的定义(3)需要和“无干扰”定义一起予以说明：

(2) await $\dfrac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \text{await B then S}\ \{Q\}}$

(3) cobegin $\dfrac{\{P_1\}S_1\{Q_1\},\cdots,\{Pn\}Sn\{Qn\}\text{是“无干扰”}}{\{P_1 \wedge \cdots \wedge Pn\}\ \text{cobegin } S_1 \| \cdots \| Sn\ \text{coend}\ \{Q_1 \wedge \cdots \wedge Qn\}}$

定义(3)表明，只要 $S_1,\cdots,S_n$ 这些进程不相互干扰，那么并行执行它们的效果如同每个进程独自执行的效果。当然，其中关键词是“干扰”两字。获得不相互干扰的一个可能的办法是不允许它们共享变量，但这个办法太局限了。一个更有用的规则是：要求在每个进程的证明 $\{Pi\}Si\{Qi\}$ 中所使用的某些断言在其它进程并行执行之后均不改变其真值。因为如果这些断言不变为假，那么证明 $\{Pi\}Si\{Qi\}$ 就仍然有效，由此，终止时 Qi 的值将仍然为真！例如，断言 $\{x \geqslant y\}$ 在执行 x := x + 1 后仍为真，而断言 $\{x = y\}$ 则不然。断定 P 在语句 S 执行之后的不变性由公式

$$\{P \wedge \text{pre}(S)\}\ S\ \{P\}$$

予以表达。下面，我们给出“无干扰”的定义。

(4) 定义：给定一证明 $\{P\}S\{Q\}$ 以及具有前置条件 pre(T) 的语句 T，如果如下两个条件成立：

① $\{Q \wedge \text{pre}(T)\}\ T\{Q\}$

② 令 S' 是 S 中的任一语句(但 S' 不能在 await 中)，有

$$\{\text{pre}(S') \wedge \text{pre}(T)\}T\{\text{pre}(S')\}$$

那么我们就说 T 不干扰 $\{P\}S\{Q\}$。

(5) 定义：如果满足以下条件：

令 T 是进程 Si 的一个 await 或一个赋值语句(它不出现在一个 await 语句中)，对于所有的 $j(j \neq i)$，T 不干扰 $\{Pj\}Sj\{Qj\}$，那么我们就说 $\{P_1\}S_1\{Q_1\},\cdots,\{Pn\}Sn\{Qn\}$ 是“无干扰”的。

在出现的正确性证明中，我们需要一次又一次地实行对正确性显然没有影响的程序变换，例如，只要赋值语句满足(1)，那么就以 S 代替 begin S end，以 x := E 代替 await true then x := E。在证明并行程序正确性之中，增加(或删去)对那些称为辅助变量的赋值，这样的变换是必要的。其中这些辅助变量仅在正确性证明以及证明其它辅助性质时方是需要的，而对于它们所在的程序而言并非必要。典型的辅助变量是用于记录程序执行的“历史”，或指出当前正在执行程序的哪一部分。在大多数情况下，唯有自己才知道辅助变量的意义。

(6) 定义：令 AV 是 S 中仅在赋值语句 x := E 出现的一个变量集合，其中 x 属于 AV，那么，AV 是 S 的一个辅助变量集合。

(7) 辅助变量变换

令 AV 是 S1的一个辅助变量集合，并且 P 和 Q 是不含有 AV 中的自由变量的断言。令 S 是把 S1中所有对 AV 中变量赋值的语句删去后而得到的语句，则

$$\frac{\{P\}\ S1\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

下面，我们将对推演系统1.(1)—1.(6)，2.(2)，2.(3)以及2.(7)的使用给出一个例子。但

首先让我们讨论一下2.(3)。规则2.(3)告诉我们要以两步来理解并行进程。首先，在研究每一进程 Si 的证明之中，要把它们看做是独立的顺序程序，而不把它们看做是完全并行执行的程序。其次，证明每一进程 Sj 的执行不干扰 Si(i≠j)。

通常，证明不干扰的方法是看进程 Sj 的执行是否与 Si 的执行发生干扰。这样，我们可以用如下词句来表达："假定 Sj 如此如此地执行，而 Si 执行这个又执行那个。"了解两个动态目标——Si 和 Sj 的交替执行是非常困难的，因而就更不可能了解多个并行进程的交替执行，且十分容易遗漏某处的讨论。

为了集中于 Sj 是否可能影响 Si 正确性的证明，我们来注意一个容易处理的静态目标。首先，我们编制一个表，在该表中列出 Si 的前置条件；而在另一表中列出 Sj 的赋值语句以及 await 语句，并且证明第二个表中的每一语句不干扰第一个表中的每一断言的真值，以此来证明不干扰就是十分机械的。

如果 Sj 的一个语句 T 干扰 Si 的一个前置条件 P，那么，或程序是不正确的，或 Si 的证明是不适当的。常常可以通过保持其证明仍然有效，来修改证明{Pi}Si{Qi}，也就是可以弱化断言，直到 Sj 不再与它们发生干扰。在任一情况下，只要程序员始终留心，就不会轻易地遗漏一个特殊情况；这个就不是以前非形式推论那样的情况。

(8) 实例研究

我们考虑并行程序设计文献中的一个标准问题。生产者进程为消费者进程生产一串值。由于生产者和消费者着手于一个变量，而它们的速度大略相等，因此在这两个进程之间插入一个缓冲是有好处的。但内存是有限的，因而缓冲的大小设为 N。我们对这个缓冲描述如下：

① buffer[0:N-1] 是共享缓冲

in = 添加到该缓冲中的元素个数

out = 自该缓冲中移走的元素个数

该缓冲含有 in−out 个值。这些值是按如下次序存放于缓冲之中：

buffer[out mod N],…,buffer[(out + in − out − 1) mod N]

在②中，我们给出了这一问题在一般环境下的解法。③是使用这一解法编制的程序，该程序把数组 A[1:M]的值复制到数组 B[1:M]中。④给出了其主程序的证明；⑤和⑥对其不同的进程给出了相应的证明。为了证明"无干扰"这一性质，首先要注意断言 I 始终是两个进程的不变式。在消费者中，唯一可能改变生产者断言真值的赋值语句是 out := out + 1；且 in − out < N 是消费者唯一可能被生产者改变真值的断言。但显然，语句 out := out + 1增加了 out 的值，使断言 in − out < N 尚为真。这样，便证明了消费者不干扰生产者；类似地可以证明生产者不干扰消费者。

② **begin** comment 缓冲描述见①；
in :=0; out:=0;
cobegin producer…
await in - out < N **then** skip;
add; buffer [in **mod** N] = next value;
markin: in :=in +1;
‖…
consumer:…

```
        await in - out > 0 then skip;
        remove: this value :=buffer [out mod N];
        markout: out := out + 1;
        ...
        coend
        end
```

③ fgl: **begin** comment 缓冲的描述见①;

```
          in := 0; out :=0; i:= 1; j:= 1;
          cobegin producer:while i ≤ M do
          begin x := A[i];
            await in − out < N then skip;
            add: buffer [in mod N] := x;
            markin : in := in + 1;
            i :=i + 1;
          end
          ‖
          consumer: while j ≤ M do
              begin
                await in − out > 0 then skip;
                remove: y := buffer [out mod N];
                markout: out := out + 1;
                B[j] := y;
                j := j + 1;
              end
          coend
        end
```

④fgl(主程序)的证明形式

```
  {M≥0}
  fgl: begin in :=0; out :=0; i:=1; j:=1;
        {I∧i=in + 1 = 1 ∧ j = out +1 = 1}
        fgl: cobegin
              {I ∧ i=in + 1 =1} producer {I ∧ i = in + 1= M+1}
              {I ∧ i=out + 1 =1} consumer {I ∧ (B[k] = A[k], 1≤k≤M)}
             coend
      end
{B[k]=A [k],1≤k≤M}
```

$$\text{其中 I}=\begin{cases}\text{buffer}[(k-1)\ \textbf{mod}\ N]=A[k],\ out<k<in\\ \qquad\wedge 0\leqslant in-out\leqslant N\\ \qquad\wedge 1\leqslant i\leqslant M+1\\ \qquad\wedge 1\leqslant j\leqslant M+1\end{cases}$$

⑤ fgl(生产者)的证明形式。其中的不变式 I 如图④中的 I

```
{I ∧ i = in + 1 }
producer:while i ≤ M do
    begin
        {I∧i=in+1∧i≤M}x:=A[i]{I∧i=in+1∧i≤M∧x=A[i]}
        await in − out < N then skip;
        {I∧i=in+1∧i≤M∧x=A[i]∧in - out < N}
        add: buffer[in mod N] = x;
        {I∧i=in+1∧i≤M∧x∧buffer[in mod N]=A[i]∧in - out < N}
        markin: in:=in + 1;
        {I∧i=in∧i≤M}i:=i+1{I∧i=in+1}
    end
    {I∧i=in+1=M+1 }
```

⑥ fgl(消费者)的证明形式。其中不变式 I 如同④中的 I。

```
{I∧IC∧j=out+1}
consumer: while j≤M do
    begin {I∧IC∧j=out+1∧j≤M}
      await in − out > 0 mod skip;
      {I∧IC∧j=out+1∧jM∧in − out > 0}
      remove: y:=buffer[out mod N];
      {I∧IC∧j=out+1∧j≤M∧in − out > 0 ∧ y=A[j]}
      markout: out:=out+1;
      {I∧IC∧j=out∧j≤M∧y=A[j]}
      B[j]=y;
      {I∧IC∧j=out+1∧B[j]=A[j]}
      j:=j+1;
      {I∧IC∧j=out+1∧j>M+1}
    end
      {I∧IC∧j=out+1=M+1}
      {I∧(B[k]=A[k],1≤k≤M)}
```

其中 IC={B[k]=A[k],1≤k<j)}

关于并行程序的正确性证明，由于引入了 await 语句，因此还应讨论封锁与死锁，即要证明程序的终止等问题[10,11]。

第十章 软件过程

1976年以前，一提及软件开发，很容易理解为就是“编程序”。随着计算机应用范围不断扩大和深化，软件越来越复杂。软件开发实践使人们认识到，软件系统开发与其它工业产品生产一样，也有必不可少的设计、制作、检验等阶段。自1976年之后，开始注重编程之前的工作，形成了“软件生存周期”这一 概念。

软件生存周期是软件产品或系统的一系列相关活动的全周期。从形成概念开始，经过开发、交付使用、在使用中不断修订和演化，直到最后被淘汰，让位于新的软件产品。根据这一概念，在软件开发中人们提出了一些软件生存周期模型，例如瀑布模型，用以划分若干个相互区别又彼此联系的阶段(活动)，组织、指导软件开发。

经过一段时间的实践，人们发现像瀑布模型那样划分阶段有许多缺点，随之又相继提出了演化模型、螺旋模型以及喷泉模型等。接着又发现提高软件生产率和软件质量的困难之处，在于开发和维护中的支持和管理问题。于是自80年代初，对软件过程进行了一系列的研究和讨论。在此基础上，IEEE 标准化委员会于1991年9月制定出“软件生存周期过程开发标准”，接着ISO/IEC 于1994年制定出“软件生存周期过程”标准。

软件过程是软件生存周期中的一系列相关过程，又称为软件生存周期过程。过程是活动的集合，活动是任务的集合，任务是将输入变换为输出的操作。活动的执行可以是顺序的，可以是重复的，可以是并行的，也可以是嵌套的。

软件过程的研究主要针对软件生产和管理。为了获得满足工程目标的软件产品，不仅涉及工程开发，还要涉及工程支持和工程管理。也就是说，软件过程不仅要有工程观点，还要有系统观点、管理观点、运行观点和用户观点。

软件过程有多种分类方法。按照不同人员的工作内容来分，软件过程可分为三类：基本过程、支持过程和组织过程。并且，当把软件过程运用到相关组织，运用到具体应用领域或具体项目时，可以根据特定情况，对各种过程和活动进行剪裁，形成所需要的软件过程模型，这就是剪裁过程。

10.1 基本过程

基本过程是指那些与软件生产直接相关的过程，包括获取过程、供应过程、开发过程、运行过程和维护过程。

1. 获取过程

获取过程是获取者为了得到一个软件系统或软件产品所进行的一系列活动。它从确定获取该系统或软件产品的需求定义开始，经过招标准备，合同的准备和修改，对供应方的监督，直到验收完成方告结束。

2. 供应过程

供应过程是供应者为获取者提供软件产品的一系列活动。它从理解系统或软件产品的需

求开始，经过投标准备，签订合同，制定计划，实施和控制，评审和评价等活动，直到交付完成。

3. 开发过程

开发过程是软件开发者所从事的一系列活动，其目的是依据合同成功地开发并交付软件。当要开发新的系统，或对已有的系统进行版本升级，以及对已有系统进行有开发活动的移植时，都要涉及开发过程。开发过程包括的活动有需求分析、设计、编码、集成、测试、安装以及验收支持等。在开发过程中，还贯穿了其它软件过程的实施：通过管理过程管理开发过程，通过剪裁过程剪裁开发过程的活动，通过改进过程参与开发过程的管理，通过基础设施过程建立基础设施，通过支持过程进行文档编制、联合评审和项目审计等。

具体地说，开发过程所涉及的活动有：

(1)过程的实施准备

这一活动的目的是为开发过程准备基本的约定。其主要任务有：

① 依据合同及软件或系统的特点，选择以下(2)到(13)的活动，所选择的活动可重复或相关联，亦可循环交替；

② 制定本过程计划，计划中至少应包含所需的内部标准、方法、工具、行为、责任以及所使用的程序设计语言等；

③ 指定各种文档的编制方式，安排其它各支持过程的实施方法(参见支持过程)。

(2) 系统需求分析

这一活动的目的是根据合同，分析系统的具体应用意图，完成系统需求规格说明书。系统需求规格说明书的内容有：

① 对系统功能和性能的需求，包括安全、保密性；

② 人机界面、操作和维护需求；

③ 设计方面的限制和对合格的需求等。

检查该系统需求规格说明书：是否能跟踪获取过程得到的系统需求并与之保持一致(参见获取过程)；是否具备可测试性；是否具备系统结构设计的条件；操作与维护是否可行等。

(3) 系统结构设计

这一活动的目的是建立一个高层的系统体系结构。该体系结构将系统需求按硬件、软件、人工操作这三个要素分为硬件配置项、软件配置项和人工操作项。

检查该结构是否达到以下要求：是否能跟踪系统需求并保持一致；所使用的方法是否符合标准；所确定的软件配置项需求是否可行；操作和维护是否可行等。

(4) 软件需求分析

这一活动的目的是确定软件需求及质量特性需求，并完成软件需求规格说明书。软件需求规格说明书的内容包括：

① 功能和性能需求；

② 外界与软件(即软件配置项)的接口；

③ 合格需求；

④ 安全需求，包括与操作、维护、环境等对人员的伤害有关的说明；

⑤ 保密需求；

⑥ 人机工程和人机界面需求；

⑦ 数据定义和数据库需求；

⑧ 现场安装及验收需求；

⑨ 用户文档；

⑩ 用户操作和运行需求；

⑪ 用户维护需求；

等等。

检查该软件需求：是否能跟踪系统需求和系统结构；是否从外部与系统需求保持一致；软件需求内部是否具备一致性；测试覆盖程度是否达到要求；是否具备可测试性；操作和维护的可行性如何；等等。

(5) 软件体系结构设计

这一活动的目的是将软件需求规格说明书转化为一个软件体系结构。其主要任务有：

① 定义并描述该结构的顶层部分；

② 为软件外部接口、数据库、软件各部件之间的接口作顶层设计；

③ 设计用户手册的初级版本；

④ 定义初步测试需求并制定软件集成计划；

等等。

检查该结构是否能跟踪软件需求，并在外部与其保持一致；结构内各部件是否具备一致性；所用设计方法与标准是否一致；是否具备进一步详细设计的条件；操作与维护是否可行；等等。

(6) 软件详细设计

这一活动的目的是详细设计软件体系结构中的每个部件。主要任务有：

① 尽可能地将每个部件细划为可进行编码、编译及测试的软件单元；

② 详细设计软件的外部接口、软件各部件之间的接口、各单元之间的接口；

③ 详细设计数据库；

④ 充实用户手册；

⑤ 定义单元测试需求并制定单元测试计划；

⑥ 充实集成测试需求并制定集成测试计划；

等等。

检查该详细设计是否能跟踪软件需求；是否在外部与体系结构具备一致性；各部件以及各单元之间是否具备一致性；所使用的设计方法是否符合标准；是否可测试；是否具备可操作性和可维护性等。

(7) 软件编码和测试

这一活动的主要任务有：

① 根据详细设计结果进行编码，提供测试数据，完成单元测试；

② 充实软件集成测试需求和集成计划；

③ 充实用户手册等；

等等。

检查本活动是否能跟踪软件需求和软件设计，并在外部与其保持一致；在软件部件内部，各单元需求之间是否具备一致性；单元的测试覆盖程度是否达到要求；编码方法是否符合标

准；是否具备软件集成及测试的条件；是否具备可操作性和可维护性；等等。

(8) 软件集成

这一活动的目的是制定计划，并将上述活动得到的各软件单元、软件部件集成为所需软件。主要任务有：

① 制定计划，计划中至少包括测试需求、步骤、数据、责任和时间进度表等内容；

② 按计划进行软件的集成；

③ 充实用户手册；

④ 针对合格需求制定合格测试集及步骤；

检查本活动以及集成好的软件是否能跟踪系统需求，并在外部是否与之保持一致性；内部是否具备一致性；测试所用的方法是否符合标准；是否符合预期结果；是否具备软件合格测试的条件；是否具备可操作性和可维护性；等等。

(9) 软件合格测试

这一活动的目的是完成软件的合格测试。依据合格要求进行合格测试，并充实用户手册。

检查本活动软件需求的测试覆盖度是否达到要求；是否符合预期结果；系统集成和测试是否可行；是否具备可操作性和可维护性等。若可能的话，应支持审计工作，审计后应准备一套完整的软件，交付给活动(10)至活动(13)。

(10) 系统集成

这一活动的目的是将交付的软件集成到系统中。主要任务有：

① 将软件同有关硬件、人工操作项以及其它必要系统集成为一个系统；

② 为系统合格测试进行必要的准备，包括开发测试集和测试步骤。

检查集成完的系统：系统需求的测试覆盖程度是否达要求；所用测试方法是否符合标准；是否符合预期结果；是否具备系统合格测试的条件；是否具备可操作性和可维护性；等等。

(11) 系统合格测试

这一活动的主要任务即依据合格要求和合格测试计划进行系统合格测试。

检查本活动：系统需求的测试覆盖是否达到要求；是否符合预期结果；是否具备可操作性和可维护性；等等。若可能的话，应支持审计工作，并在审计结束后准备一套完整的软件以便进行软件安装或软件验收支持。

(12) 软件安装

这一活动的目的是在目标环境中安装软件。主要任务：

① 制定安装计划，包括与软件安装有关的信息和资源；

② 实施软件安装，注意保证该软件和数据库能按合同的规定初始化、运行和终止。

(13) 软件验收支持

这一活动的目的是支持获取者对软件的验收评审和测试。主要任务有：

① 支持验收评审和测试；

② 按合同交付文档及代码；

③ 依据合同向获取者提供培训和支持。

4. 运行过程

运行过程是用户和操作人员在用户的业务运行环境中为了使系统或软件产品投入运行所进行的一系列有关的活动。此过程的目的是在软件开发过程完成后，将该系统从开发的环境转

移到用户的业务运行环境中运行；在运行时对用户的要求提供帮助和咨询；并对运行效果作出评价。此过程可为开发过程和维护过程提供有关的反馈信息，作为改进系统的依据。运行管理者可根据软件项目的总要求按照软件管理过程（参见“软件管理过程”）的内容对运行过程进行管理。

运行过程一般包括以下7个活动：实施过程的准备；运行测试；系统到业务运行环境的转移；系统运行；对用户运行的支持；系统运行评价；用户运行评价等。

① 在开始准备时，要了解清楚软件开发过程的结果、准备用户的业务运行环境、制定运行过程的实施计划和运行评价标准。实施计划包括任务的分配、过渡计划（例如人机并行运行考虑）、操作培训计划、运行步骤等。

② 为了使系统转移到用户的业务运行环境中运行，操作者必须在业务运行环境中对系统进行运行测试。如果其测试结果能满足运行的基准要求，则可将系统进行交付。

③ 在系统向业务运行环境转移中，必须进行以下工作：编制有关转移的文档；进行数据的转移；进行程序的转移；进行试运行；在试运行中进行跟踪；进行业务转移。当这些工作都完成后，进行转移的确认，并将转移结果的评价通知开发人员。

④ 系统投入正式运行后，必须进行管理。运行管理者和操作者应当收集运行的数据，判别、记录和解决运行中发现的问题，并改进运行环境。

⑤ 操作（开发）者应为用户提供运行上的支持，包括对用户进行操作培训和其它培训；应建立接收、记录和解决用户请求的步骤；对用户的请求提供帮助和咨询服务，并对这些请求的响应进行记录和监控；必要时，操作（开发）者应将用户的请求向软件维护过程（参见“软件维护过程”）提供改进信息或修改请求，并由软件维护过程进行解决。

⑥ 系统运行评价主要是指对系统的质量指标和其它方面评价。例如系统的响应分析、系统的运行效率、系统的安全性和保密性、系统故障分析、数据及媒体的管理以及人员管理、系统管理、运行时间管理等。

⑦ 用户运行评价包括用户所要求的业务功能的实现程度、使用系统的容易程度、用户所设置的资源的运行和管理；系统运行效果以及用户对系统的改进要求等。

5. 维护过程

维护过程是软件维护人员所从事的一系列活动，其目的是在保持软件整体性能的同时修改它，使它达到某一需求，直到其退役才告终止。从维护方式上讲有三种维护：改正性维护、适应性维护以及完善性维护。诊断并校正一个或多个错误的活动称为改正性维护。为了适应变化的环境对软件修改的活动称为适应性维护。当软件投入运行后，根据用户新的需求，或增加新的能力，或改进现有功能所进行的活动称为完善性维护。

维护过程包含的活动有：问题分析和修改分析、修改/实施。对维护的评审/验收、移植、软件退役等。维护过程同时还贯穿软件过程中其它过程的实施；维护人员通过管理过程管理维护过程（参见管理过程）；通过剪裁过程剪裁维护过程中的活动（参见剪裁过程）；通过改进过程参与维护过程的管理（参见软件过程）；通过支持过程中的各个过程实施维护过程中的文档编制、评审、质量保证等（参见支持过程）。当进行某个活动时，维护过程可能需进入开发过程（如在实施软件的修改时），这时维护人员即是那里的开发人员（参见开发过程）。

具体地说，维护过程所涉及的活动有：

（1）过程的实施准备

这一活动的目的是为维护过程准备最基本的约定。其主要任务有：

① 制定(2)到(6)的活动实施计划和步骤；

② 确定对用户的问题报告和修改请求进行接收、跟踪的步骤以及向用户反馈信息的方式和步骤；

③ 指定文档编制方式、配置管理方式以及各支持过程的实施方法。

(2) 问题分析和修改分析

这一活动的目的是分析软件修改将对系统、接口系统带来的影响并确立修改方案。主要任务有：

① 分析维护类型，是改进型、改正型，还是适应新环境型的维护；分析维护的范围，包括修改规模、成本、时间；分析维护对关键问题(如性能、保密性)的影响；

② 在分析的基础上，选择实施修改的方案。

(3) 修改的实施

这一活动的目的是对问题及修改进行更为详细的分析，并实施修改。主要任务有：

① 详细分析问题报告和修改请求，决定哪些文档、软件单元和版本需要修改；

② 实施修改。此时维护人员需进入开发过程完成修改(参见开发过程)。开发过程中的需求应作出相应的修改，最低要求是保证未经修改的需求不受影响，而新的修改过程的需求得到完全、正确的实现。

(4) 对维护进行评审/验收

这一活动的任务是维护人员同授权修改的机构一起进行评审、评定经过修改之后系统的整体性能。

(5) 移植

这一活动的目的是将一个系统或软件从一个旧的操作环境移植到一个新的操作环境中，并保证该移植活动的正确性。由于涉及软件的修改，因而也是维护活动。其主要任务有：

① 制定移植计划并执行该计划。计划中至少包括对需求分析和移植的定义；移植工具的开发问题；软件和数据的转换问题；移植计划的执行问题；移植的验证问题；以后对旧环境的支持问题；等等：

② 向用户通告移植计划和移植执行情况；

③ 旧环境和新环境最好并行运行，并向用户提供必要的培训；

④ 对移植活动及移植后的结果进行评审。

(6) 软件退役

软件将根据所有者的要求予以退役。此时，维护人员应当：

① 制定软件支持的撤销计划，并执行该计划。计划中至少包括在多长时间后全部或部分地停止软件支持；系统及有关文档如何存档；若以后仍需要支持时的责任问题；若需要，转移到新的软件上的问题；

② 向用户通告退役计划及执行情况；

③ 将退役软件的所有文档、记录数据归档。

10.2 支持过程

支持过程是有关各方按他们的支持目标所从事的一系列相关活动。支持过程有助于提高系统或软件产品的质量，有助于它们的满意运行。这类软件过程可被软件生存周期中其它类软件过程（如获取过程、供应过程、开发过程、运行过程、维护过程等）或本类中的其它软件过程所使用。软件过程的各活动均可使用支持过程。支持过程可由使用它们的组织来实施；或作为一种服务，由一个独立的组织来实施；也可作为项目的一项规定内容，由客户来实施。一个支持过程中的活动，可由实施该过程的组织负责。这类软件过程包括文档过程、配置管理过程、质量保证过程、验证过程、确认过程、联合评审过程、审计过程、问题解决过程等。

10.2.1 文档过程

文档过程是一个记录由某一过程或活动所产生的信息的过程。这一过程的作用是建立计划、设计、开发、制作、编辑、发行和维护等各类文档。这些文档对系统或对软件管理者、工程师以及用户都是必需的。文档过程由以下一些活动构成：

（1）过程的实施准备

这一活动的主要任务是制订一个文档编制计划。该计划确定需产生的所有文档。对于每一个文档，应列出其标题或名称、目的、预期的使用者、制作过程和各类参加人员的责任以及制作的计划进度等。

（2）设计与开发

这一活动的主要任务有：

① 根据适用的文档标准，对确定的每一文档，设计其格式、内容说明、图表设置以及包装等；

② 应保证各文档输入数据的来源和适合性；

③ 应对所编制的文档的格式、技术内容以及表达方式是否符合文档标准进行审查。在分发前需经主管人员批准。

（3）制作与发行

这一活动的主要任务有：

① 对文档进行制作和包装，并按计划提供给预期的使用者。主要材料的存放应考虑到项目的记录、安全、维护和备份；

② 应按照配置管理过程建立控制。

（4）维护

当对现存的软件产品进行修改时，需要实施维护过程中的有关任务。对处于配置管理之下的软件产品的修改，应按照配置管理过程进行管理。

10.2.2 配置管理过程

配置管理过程是一个应用管理和技术步骤来完成下列工作的过程。这些工作包括确定、定义一个系统中的软件配置项和基线；控制配置项的修改与交付；记录和报告配置项的完成情况和修改请求；保证配置项的完整性、相容性和正确性；以及控制配置项的存储、处理和提交。配

置管理过程由以下一些活动构成：

（1）过程的实施准备

这一活动的主要任务是 制订配置管理计划。该计划应指明各项配置管理活动；实施这些活动的步骤；负责实施各项配置管理活动的机构；该机构与其它机构（如软件开发机构）的关系。

（2）配置的确定

这一活动的主要任务有制订确定项目要控制的配置项及其各版本的方案。对每一配置项其各版本应指明建立基线的文档；存放该文档的媒体；参照版本；等等。

（3）配置的控制

这一活动的主要任务是修改请求的确定和记录；修改的分析与评价；批准或不批准请求；修改项的实施、验证和交付。应保存审计记录以跟踪每一修改、修改的原因和修改的授权。

（4）配置情况报告

这一活动的主要任务是建立指出所控制的项（包括基线）的现状和历史的管理记录以及情况报告。情况报告应包括项目的修改次数、配置项的最新版本、发行标识、发行号、各次发行的比较。等等。

（5）配置的评价

这一活动的主要任务是决定和保证各配置项对其需求而言的功能完整程度和各配置项的物理完整程度（其设计和编码是否反映最新的技术描述）。

（6）发行管理与提交

这一活动的主要任务是控制软件和文档的发行和提交。其中主要的代码和文档应妥善保管。涉及安全或保密的关键功能的代码和文档，应按有关机构的政策处理、存储、包装和提交。

10.2.3 质量保证过程

质量保证过程是一个为使软件过程和软件产品符合所规定的需求，并按所制订的计划完成提供适当保证的过程。为了避免产生偏见，实施质量保证的人员不能是直接负责软件产品开发的人员，并应在组织上给予独立的权限。质量保证过程由以下一些活动构成：

（1）过程的实施准备

这一活动的主要任务有：

① 根据项目的具体情况剪裁质量保证过程，以实现质量保证的目的——保证软件产品和为提供这些产品所使用的过程符合规定的需求并按计划完成；

② 制订进行质量保证过程各项活动和任务的计划，把它编成文档并实施，在合同期内加以维护。

质量保证过程应和相关的验证过程、确认过程、联合评审过程和审计过程密切配合。

（2）软件产品的质量保证

这一活动的主要任务有：

① 保证合同所需的各项计划均已编制成文档，内容符合合同要求，互相一致，并保证这些计划的实施；

② 保证软件和有关文档符合合同要求并按计划完成；

③ 在准备软件产品的提交时，应保证软件产品完全满足合同要求且为用户所接受。

(3) 软件过程的质量保证

这一活动的主要任务有：

① 保证项目实施的供应、开发、运行、维护和支持(包括质量保证)过程符合合同要求并按计划完成；

② 保证开发单位或组织内部的软件工程实践、开发环境、测试环境、资料等的适合且与合同要求相一致；

③ 保证用户及其它合作伙伴按合同和计划提供必要支持和合作；

④ 保证软件产品和过程的度量符合所建立的标准和步骤；

⑤ 保证项目组成员具有参加项目所必需的知识和技能，并接受必要的培训。

10.2.4 验证过程

验证过程的目的是确定一个系统或软件的需求是否完备和正确，以及每一阶段的软件产品是否达到了前面各阶段对它提出的要求或条件。验证可以和使用它的过程(如供应过程、开发过程、运行过程或维护过程)结合在一起实施，也可由一个独立的机构来实施。

验证过程由以下一些活动构成：

(1) 过程的实施准备

① 根据项目的范围、大小和复杂性，指明需要验证的活动和软件产品，并选取相应的验证活动及其实施的方法、技术和工具；

② 根据上述指明的验证活动制订验证计划，编成文档，并组织实施；

③ 把验证所发现的问题或不一致现象，作为问题解决过程的输入。验证活动的结果应能被获取者和其它有关机构所利用。

(2) 验证

① 合同验证：主要验证供应者满足需求的能力，需求是否完备、一致和覆盖用户的要求，是否已规定处理需求变更的适当步骤，是否已规定合作伙伴间的界面与协作的步骤和范围，是否已规定验收准则和步骤等；

② 过程验证：主要验证项目计划需求是否已合适和适时，为项目选取的过程是否合适、按计划完成并满足合同要求，用于项目的标准、步骤和环境是否合适，人员是否按合同要求进行了培训等；

③ 需求验证：主要验证系统需求的完备性、相容性、正确性、可行性和可测试性，系统需求是否已适当地分配到各硬件配置项和软件配置项，软件需求的完备性、相容性、正确性、可行性、可测试性以及是否精确地反映了系统需求，与安全性、保密性和关键性有关的软件需求是否正确等；

④ 设计验证：主要验证设计是否正确并且是否符合可跟踪需求，设计是否实现了适当的事件序列、输入、输出、界面、逻辑流程、错误恢复等，所选的设计是否来自需求，设计是否正确地实现了安全性、保密性和其它关键性需求等；

⑤ 代码验证：主要验证关键的代码是否可跟踪设计和需求、可测试、正确、符合需求和编码标准，代码是否实现了适当的事件序列、输入、输出、界面、逻辑流程、错误恢复等，所选的代码是否来自设计或需求，代码是否正确地实现了安全性、保密性和其它关键性需求等；

⑥ 集成验证：主要验证每一个软件配置项的部件和单元是否已完全、正确地集成到该软

件配置项中，系统的部件（硬件配置项、软件配置项、人工操作）是否已完全、正确地集成到该系统中，集成的各项任务是否均已按集成计划完成等；

⑦ 文档验证：主要验证文档是否合适、完整和相容，文档的制作是否及时，文档的配置管理是否遵照所规定的步骤等。

10.2.5 确认过程

确认过程是确定需求和最终建成的系统或软件是否满足原计划特定应用的过程。确认可由一个独立于供应人员、开发人员、操作人员或维护人员的机构来执行。确认过程由以下一些活动构成：

（1）过程的实施准备

这一活动的主要任务有：

① 应开发确认计划并编成文档。该计划至少应包括要确认的配置项，要实行的确认任务，确认的资源、责任和计划进度，向获取者和其它合作伙伴提交确认报告的步骤等；

② 应组织实施确认计划。对于在确认中发现的问题和不一致现象，应作为问题解决过程的输入。确认活动的结果应能被获取者和其它有关机构所利用。

（2）确认

这一活动的主要任务有：

① 准备测试需求、测试用例和用于分析测试结果的测试规格说明书；

② 确认这些测试需求、测试用例和测试规格说明书能反映该特定应用的特殊要求；

③ 按上述要求实施测试；

④ 确认该软件所规定的衡量标准；

⑤ 确认该软件实现了安全性、保密性和其它关键性需求。

10.2.6 联合评审过程

联合评审过程是评价项目的某个活动或阶段的执行情况和产品是否合适的过程。它可以由任意两个合作伙伴所使用，由其中的一方评审另一方。联合评审过程由以下一些活动构成：

（1）过程的实施准备

联合评审过程按以下要求进行：

① 应按项目计划的规定定期执行；

② 需评审的资源应经双方同意；

③ 每次评审前双方应就会议日程、要评审的软件产品、范围与步骤等取得一致；

④ 评审中所发现的问题应加以记录并作为问题解决过程的输入；

⑤ 评审结果应归档并分发。评审方要把评审结果的合适性（如批准、不批准）通知被评审方。

（2）项目管理评审

主要任务是对照项目的计划、进度表、标准和指导方针等来评价项目的进展情况。

（3）技术评审

主要任务是评价软件产品的完备性和适合性及其和标准和规格说明书的一致程度。

10.2.7 审计过程

审计过程的目的是确定遵照需求、计划合同的程度。审计可由任何两个合作伙伴使用，由其中一方审计另一方的软件产品或活动。审计过程由以下一些活动构成：

(1) 过程的实施准备

审计过程按以下要求进行：

① 应按项目计划的规定定期执行；

② 审计人员不能对他们审计的软件产品和活动有任何直接责任；

③ 每次审计前双方应就日程、要审计的软件产品、范围与步骤等取得一致；

④ 审计中所发现的问题应加以记录并作为问题解决过程的输入；

⑤ 审计结果应归档并提供给被审计方。

(2) 审计

审计的内容包括：

① 软件产品是否反映了设计文档的要求；

② 文档所描述的验收评审和测试需求是否适合于软件产品的验收；

③ 测试数据是否遵照规格说明书的要求；

④ 软件产品是否测试通过并满足规格说明书的要求；

⑤ 测试报告和使用手册是否完整和适合；

⑥ 各项活动是否都已按可应用的需求、计划和合同完成；

⑦ 成本和进度表是否符合于所制订的计划。

10.2.8 问题解决过程

问题解决过程是一个用于分析和排除在开发、运行、维护或其它过程中发现的问题或不一致(不管其性质和来源)的过程。其目的是提供一种适时的、可信赖的并编成文档的手段，以保证分析和排除所有的问题并指明各种倾向。问题解决过程由以下一些活动构成：

(1) 过程的实施准备

问题解决过程应是一个保证做到以下各点的闭环：所有问题被及时报告并进入问题解决过程；启动有关活动；原因被指明、分析和消除(如有可能的话)；问题被解决；跟踪和报告实施情况；按合同规定保存问题的记录。

这一活动的主要任务有：

① 根据对问题分类及排定优先次序的方案，对每一问题进行归类；

② 评价问题是否已解决，不良倾向是否已被扭转，修改是否已在适当的过程和软件产品中正确实施，并确定是否会带来新的问题。

(2) 问题解决

当在一个软件产品或活动中发现问题或不一致时，应对所发现的每个问题编写问题/更改报告。问题/更改报告应描述需解决的问题及其原因(如可能的话)。该报告将作为问题解决过程的输入去纠正缺陷、产生缺陷的原因，以及扭转不良的倾向。

10.3 组织过程

组织过程是指那些与软件生产组织有关的过程，包括管理过程、基础设施过程、改进过程和培训过程。

1. 管理过程

管理过程是软件生存周期过程中管理者所从事的一系列活动。管理者负责对所从事的过程，例如获取、供应、开发和支持等过程的活动进行管理；软件管理过程适用于必须对各自的过程进行管理的任何一方。

软件管理过程的目的是在一定的时间和预算范围内，有效地利用人力、资源、技术和工具，完成预定的系统或软件产品，实现预定的功能和其它质量目标。管理技能往往是系统或软件产品成败和软件质量高低的重要条件。大型软件项目涉及较多的人力、资源和时间，管理问题尤为突出。

软件生产是一项劳动和智力密集的活动，它是以人为中心的过程，具有可见性差和定量化难的特点。可见性差是指软件研制进度不易标识，存在问题不易及时发现和纠正，其过程容易出现修改和反复。定量化难指软件的成本、生产率和质量不易度量。因此，软件管理有特殊的复杂性。

软件管理过程是随着软件工程的发展而发展的。早在60年代末，已经有人讨论大型软件研制项目的组织管理问题。软件工程实践中发生的种种问题，往往是与管理密切相关的。例如，进度推迟、经费超支、质量差、程序人员不称职等。可管理性成为软件生产工程化的重要标志。因此，与软件工程化、产品化过程相适应，形成新的分工，出现了组织管理软件生产的管理员。这些管理员的管理活动体现了最初的软件管理过程。

软件管理活动的研究与软件管理的研究密切相关。最初主要在人员和组织方面，如软件心理学的研究。随着软件技术的发展，软件管理本身出现了专门的方法、技术和工具，来支持软件管理活动。

80年代以来，随着软件和软件工程的进一步发展，例如软件工程模型化、标准化、软件质量度量及评价技术、软件经济学、软件开发环境和工具等方面的发展，特别是1984年以来软件过程的研究，明确了软件管理过程的概念和内容，进一步促进了软件管理过程的发展。

软件管理的对象是进度、系统规模和工作量估计、经费、组织和人员、风险、质量、作业、环境配置等。因此，软件管理一般可分为进度管理、成本管理、质量管理、人员管理、资源管理和标准化管理等。这些管理一般都有其各自的活动内容。软件管理过程把这些活动归纳起来，抽取其活动共性，一般可包含下述活动：

(1) 过程的实施准备

一开始先要搞清楚进行管理过程的需求，管理者通过调查研究确认达到需求的可能性。在必要时可通过有关各方协商来修订和完善需求。

(2) 管理计划的制定

制定计划的任务包括规定进度、分配资源、决定项目有关的组织和承担人员(包括人员的地位、作用、职责、规章制度等)、根据规模和工作量估计进行分配任务、风险定量化、制定质量管理指标、编制预算和成本、准备环境和基础设施等。

(3) 计划的实施和控制

管理者根据需求对过程进行控制。他们监督过程的实施、提供过程进展的内部报告和按合同规定向获取方提供外部报告，并应当调查、分析和解决在执行过程中发现的问题，对计划进行调整和修改。问题及其解决办法都应写成文档。

(4) 计划完成程度的评审和评价

管理人员应对计划完成程度进行评审，对项目进行评价，并对计划和项目进行检查，使计划和项目在完成或变更之后保持完整性和一致性。

(5) 管理过程完成时编写文档

管理者根据合同确定此过程是否完成。如果完成，应从完整性方面检查项目完成的结果和记录，并把这些结果和记录编写成文档和存档。

2. 基础设施过程

基础设施过程是建立、维护任何其它过程所需的基础设施的过程。基础设施可以包括硬件、软件、工具、技术、标准以及开发、运行、维护所需的设施。软件过程中的许多过程都应明确该过程的基础设施。本过程的主要活动是定义并建立各过程所需的基础设施，并在其它相关过程执行时维护所建立的基础设施。

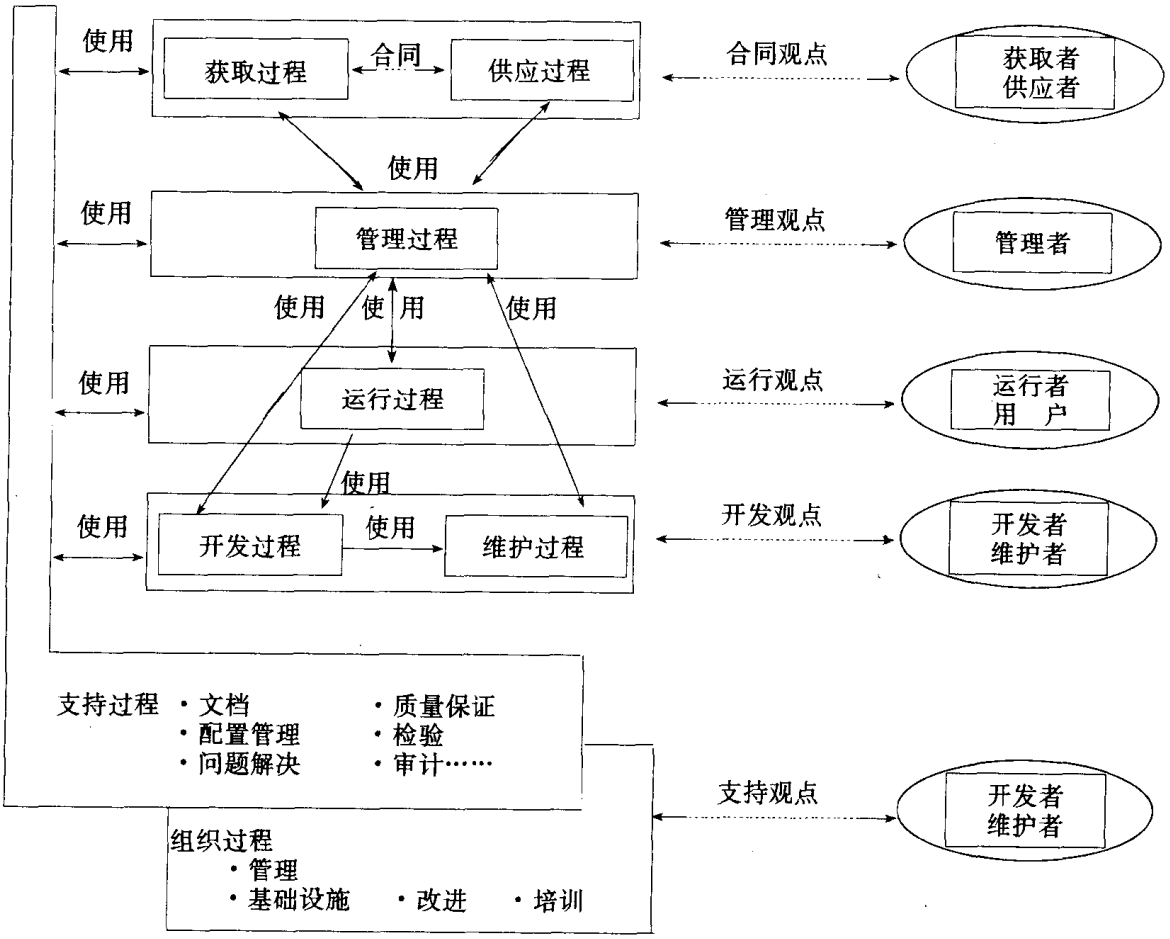


图10.1 软件工程过程及其关系

3. 改进过程

改进过程是建立、评估、度量、控制和改进软件生存周期过程的过程。其主要活动有制定一套组织计划，评估相关过程并实施分析、改进过程。

软件过程中的任何一个过程都有可能涉及这一过程。

4. 培训过程

培训过程是为系统或软件产品提供人员培训的过程。软件获取、开发、运行和维护的效果主要取决于有关人员所具备的知识和熟练程度。因此，拟定人员培训计划并及早实施是非常必要的。本过程要完成的主要活动有制定所需的人员计划及培训计划，开发培训资料及实施培训活动等。

软件生存周期过程及其关系如图10.1所示。

10.4 剪裁过程和过程模型建造技术

10.4.1 剪裁过程

为了有效地实施软件过程，应针对特定领域的软件工程，对选定的过程模型和标准进行剪裁，以形成这一工程的模型及标准，形成该工程的各个软件过程和活动。

剪裁过程作为一类软件过程，是对软件过程和活动实施剪裁的过程。其主要活动有：

1. 指明工程环境

这一活动应指明影响剪裁的工程环境特征，如使用的工程模型和方法，系统和软件需求，机构的政策与策略，系统和软件的规模、重要性和类型，参与工程的人员和合作伙伴的素质、数量等。

2. 收集信息

一般来说，参与剪裁的人员包括用户、工程支持人员、负责合同签订的人员以及潜在的投标者，应向所有会影响剪裁的组织收集信息。

3. 选取过程、活动和任务

这一活动将根据上述收集的数据，确定要实施的过程、活动和任务。

4. 编制文档

这一活动为所有剪裁决定及这些决定的理由编制文档。

剪裁可分为两级，第一级是根据应用领域的不同进行剪裁，第二级是根据每一具体项目或合同进行剪裁。就第一级而言，涉及了开发过程的剪裁。对嵌入或集成在系统中的软件，应考虑软件剪裁过程的所有活动，并且必须明确开发者是否实施或支持系统的活动；对于独立的软件，关于系统的活动可以是不需要的，但应加以考虑。也涉及了与评价有关活动的剪裁。评价可分为五类，即过程内部的评价，验证和确认，联合评审和审计，质量保证以及过程改进。应根据项目或机构所涉及的范围、大小、复杂性和重要性相应地选取和剪裁这些评价。在实施剪裁过程中，还应指明或考虑一些关键的项目特征，例如机构政策、获取策略、支持的概念、生存周期模型、合作伙伴、系统级特征、软件级特征以及风险性等。

10.4.2 过程建模技术简介

过程建模技术是当前软件工程学科的一个重要研究方向，涉及建模目的、方法及描述等内

容。为了对建模目的有一个比较好的理解，首先简述一下软件过程具有的主要性质。

1. 软件过程主要性质

(1) 分工协作性

一个软件项目通常是由多个人参与研究，这些人又有从事管理和从事开发工作之分；另外，就开发工作而言，又可以把参与人员分为系统开发组、测试组等不同活动的工作组，每组由多人分工负责不同的任务，从而形成一个多层次多侧面的项目组织关系。软件项目的成功与否主要依赖这种个人之间、小组之间及各类参与人员之间的密切合作。

(2) 易变性

软件活动由许多活动组成，这些活动在某一时刻的状态是事先难以确定的。由于用户需求经常变化，使软件过程在执行中可能增加一些活动，也可能对某项任务需要进一步迭代执行。这些都取决于任务完成情况和评审结果，这些因素是预先无法准确估计到的。

(3) 动态性

软件过程具有很强的时间依赖性，即软件过程本身的状态随着开发活动的进展而变化。例如，在某一活动中的角色，可能在另一活动中担任另外的角色，而某一活动的输出产品可能是另一活动的输入。

(4) 逐步细化性

软件过程的高度复杂性使人们无法在事先就对过程的所有细节都了解得很清楚，也没有必要这样做。因此，开始时只能对过程进行较粗的计划。只有到了一定阶段，才能真正详细地计划某一过程。

(5) 不可完全形式化

软件过程中包括大量的创造性活动，一般来说，这些活动是不可形式化的，也不必形式化。对创造性较强的智能活动，只需给出有关的约束条件，其余工作可由开发者自由的发挥。

(6) 并行性

软件过程是由多人或多个小组分工协作完成的，所以在某个时间段内就会有多人并行地完成某些任务，这是软件过程的一个重要特征。

2. 软件过程建模技术综述

软件过程建模是对实际的软件过程的再工程(Reengineering)，即在简化的实际过程基础上对软件过程进行抽象描述。过程建模活动是软件过程中最主要的活动，所有其它工程活动都是基于建模活动的结果来进行的。过程建模技术包括四方面内容：建模目的、建模方法、建模语言和软件过程与过程模型的度量。其中建模目的决定了建模活动的范围；建模方法和建模语言是对建模目的的支持和过程模型形式化的手段，并决定了过程模型的质量；对过程模型的度量与评估是修改和演化过程模型的基础，以便使过程模型更好地与软件过程相吻合，并充分反映一个组织的过程成熟度。可见，过程建模方法和建模语言是过程建模技术的关键。

(1) 过程建模目的

过程建模极为广泛，综合起来主要有：

① 使人们易于理解软件过程并在此基础上进行交流

由于软件过程的复杂性，使人们常常不能全面、仔细地了解过程的全貌，况且背景、经历不同的人对同一过程可能会有不同的理解。因此通过过程建模，可以使不同的人对过程达到一个共识，共享过程知识，并在此基础上进行交流。

② 支持对过程的分析

通过对过程进行形式化或非形式化的建模，为进行过程分析提供了基础。人们可以对过程活动以及活动间相互关系进行分析、比较和预测，以评估和改善过程的有效性。

③ 支持过程中的通信

由于软件过程多是由多人在一定时间阶段内执行的，所以过程中涉及到的人员之间存在大量的信息交流活动。过程模型可以为有关人员、小组和项目提供必要的过程信息和通信支持，使人们之间协同工作更加有效。

④ 支持过程的改进

可以对过程模型进行分析，使人们识别过程的各个部分的功效，找出其可能的改善部分，对模型进行修改；并在模型实际修改前通过分析模型中各部分之间关系来估计出修改后产生的影响。从而支持严格管理下的过程进化，并使软件组织积累了有效的过程改善知识。

⑤ 支持过程的管理

过程模型可以帮助管理人员制定项目计划，监控、管理和协调项目实施过程，估算软件创建或演化的属性。例如，估算各活动的实际进展情况。过程模型还可作为过程度量的基础，通过它来定义度量点、度量内容等。

⑥ 支持过程复用

通过过程模型可以重用良好定义的软件过程，这种复用可以在不同的项目之间进行。还可以针对具体项目的特点，对已有的过程进行裁剪、扩充，使之适合特定项目复用的要求。

⑦ 提供对过程的自动执行支持

这种对软件过程的自动执行支持需要基于一个过程驱动的软件工程环境。在这种环境中，通过过程实施机制，按照定义的过程模型自动地驱动实际过程的执行。这种支持包括对环境用户的资源配置、提供各用户之间的通信服务以及组织和协调个人或小组之间的工作等。

(2) 过程建模方法

过程建模涉及到软件产品的开发与维护、软件项目管理、过程管理与过程改善等各个方面，涉及到过程的活动、角色、产品、资源和约束等各种过程实体，还涉及到建模所用的形式化方法，加之软件过程本身的复杂多样性，因而使得构造过程模型的方法也是多种多样的。把目前已有的各种建模方法及建模形式化语言综合起来考虑，大致有两种不同的过程建模分类方法，它们是主要考虑过程所涉及的实体的分类方法和主要考虑建模所采用的不同形式化方法的分类方法。下面分别评述这两种分类方法中所包含的各种建模方法。

第一种分类方法　主要考虑过程所涉及的实体的分类方法

这种分类方法是以过程所涉及的各个过程实体(如活动、角色、产品、资源和约束等)为出发点来考虑过程建模。主要有两种不同的建模方法。

① 以活动为中心的建模方法

这种建模方法以过程中一类主要实体——过程活动为中心构造过程模型。步骤是先确定这些活动以及它们之间的执行顺序，再收集与各个活动相关的其它数据，例如活动所涉及的角色、产品、资源和约束等。这种方法能够直观地反映实际过程的工作流程，且无二义性，使人们很容易理解、分析、计划、管理和控制过程的执行。特别对管理人员来说，他们能够清楚地监视过程的进展情况，从而进行有效的协调与管理。这种方法的缺点是，由于活动是动态实体，而且实际流程也是动态的多变的，因此过程模型缺乏稳定性。即实际过程中发生任何变化都将影响

到原来的过程模型。况且在建模时就要求建模人员具有所建模之过程活动的预备知识，并要在模型实施中进行一致性维护。这种方法比较适合对实际过程的执行指导与控制等目的。支持这种方法建模的形式化语言有 APPL/A，HFSP，MELMAC，PRISM 等。

② 以角色为中心的建模方法

这种方法以过程中另一类主要实体——角色为中心建模。因为角色是组织结构中的基本构成因素，是一个易于接受和理解的相对稳定的抽象实体，并且一个项目的任务通常是按照角色来分解的，因此以角色为中心的建模方法显得比较自然。建模步骤是先确定各个角色的任务以及角色之间的耦合关系，再以角色为中心收集过程的其它数据。如活动、产品、资源和约束等等。由于角色是过程的一个不变的实体，因此构造的过程模型具有较好的稳定性。这种方法还能明确地描述过程的组织方面的信息，使得参与项目的人员易于明确自己的任务，也便于对项目的计划、管理与控制。它的缺点是不能对过程的工作流程有一个明显的描述和定义，人们难以从整体上了解一个过程和他们在过程中的位置。另外，随任务的逐步分解细化，涉及的角色也增多，角色间的关系也更复杂，不利于低层次上的过程管理。这种方法的代表有 Role Interaction Nets(Cain 93)等等。

除上述两种建模方法外，在这种分类中还有以产品为中心的建模(waters 89)和基于过程模板的建模(SPC 93)等方法，因它们都存在一些问题，在此从略。读者如有兴趣可阅读有关参考文献。

第二种分类方法　主要考虑过程建模所采用的形式化方法的分类

不论采用什么方法进行过程建模，都需要有相应的形式化方法进行支持。本节介绍主要考虑建模所采用的不同形式化方法和语言风格对过程建模方法的分类，这种分类把过程建模主要分为五种不同的方法。

① 过程程序设计方法

这种建模方法的出发点是"软件过程也是软件"的概念(Osterweil 87)。Osterweil 认为软件过程与软件产品具有广泛的类同性，对软件过程的描述亦是一种程序设计形式。这种方法是把过程所涉及的软件对象用其所需工具与开发方法编程，它通过关系、触发器和谓词等机制对过程的功能、行为和对象进行详细、确定的算法描述。这种方法不是控制活动的执行方法，而是控制生产产品的交互过程和产品的结构。它的优点是能成功地进行交互控制工作，并自动维护对象之间的一致性。它的缺点是由于其严格过程化，不能支持大范围的并发活动，且无法描述软件过程的动态变化情况。支持这种建模方法的形式化语言有 APPL/A，Hindsight 等。

② 功能分解方法

这种建模方法把一个软件过程用带有输入属性和输出属性的一个过程元素集来表示。即把一个过程定义为反映输入与输出关系的数学函数集。这些函数可以按照语法进一步进行层次分解，形成一个过程的多个子过程步。这种分解可以一直进行下去，直至产生的子过程步映射到一个外部工具或由人员操作实现。这种方法支持过程步的并行执行，以及过程步之间的串行、迭代等执行方法。它还提供了控制过程状态行为的元操作，例如创建、挂起、恢复执行等。这种支持对过程的理解、执行指导和复用等建模目的，主要面向过程的功能、行为和信息等方面。

③ 基于 Petri 网的建模方法

当前的过程建模方法中，基于 Petri 网或其变种(如扩充的 Petri 网、FUNSOFT 网等)进行过程建模的方法占有相当数量。因为 Petri 网具有很强的表达能力，能够有效地形式化描述

软件过程的并发性和活动与产品之间关系。而且这种图形表示易于理解和管理软件过程。这种方法的优点是较好地考虑了任务的激活条件、活动的执行顺序和活动产生的信息实体之间的转换情况。它的缺点是忽略了活动对内部状态所产生的影响。另外，由于网的全局相关性，使对过程的任何改动都会影响到其它部分，不利于过程复用。这种方法建模的代表有基于 FUNSOFT 网的 PRISM 项目、基于扩充 Petri 网的 MELMAC 和 SLANG，Process WEAVER 和 Role Interaction Nets 等。

④ 基于规则的建模方法

基于规则的建模方法和语言是人们普遍看好的一种过程建模技术。这种方法提供了活动的动态链接机制，从而很自然地描述了过程的不可预见性，也为人们控制过程提供了最为灵活的手段。这类语言通常还提供回溯、向前链接、向后链接等自动执行机制，以及规则推理、调度和控制过程活动的机制，并为过程的修改提供了灵活方便的环境。但是这种方法不利于人们理解软件过程，在构造、分析过程模型中也存在较大困难。它对多人并行工作和协同工作也缺乏有效的支持。这种方法的代表有 PEACE，Marvel MASP 等等。

⑤ 基于知识的建模方法

基于知识建模方法提供了对过程模型的增量式形式说明能力和可复用能力。这种方法把过程知识（例如过程活动、过程实施者、产品对象和工具以及它们之间关系等）用面向对象方法抽象成各个不同的类，存于知识库中。过程建模时，根据需要查询知识库，从中获取有关过程活动及其它成分的抽象描述，从中选取或构造所需的过程模型，并对其进行分析和推理，最后生成过程实例及相应的活动计划。这种方法中，模型的构造是分层的，由活动的类和子类构成类体系结构。体系中每个活动类或子类都对应有多种资源需求，例如要加工的数据、所需工具、开发角色等。用类与子类之间的关系描述来表示各种过程关系，如控制流关系、任务的前置和后置条件、不同角色之间的上下级关系、产品的组成关系等。这种方法的代表有 Splib（P. Mi 92）、SMART（Gary 94）等。

总的来说，一种建模方法是否合适，对其评定完全依赖于这种方法所建立的模型是否达到了建模目的。为了使建模方法具有广泛通用性和适应能力，集成多种风格形式化于一身的建模方法是十分必要的，即形成混合风格的建模方法。然而这种混合风格的方法在实际应用中还存在许多问题，例如各个模型成分之间的转换、通信、协调和交互作用等。这种混合风格建模的代表有 STATEMATE 等。

（3）过程建模语言

过程建模语言是用于构造过程模型并把它形式化的基本工具。建模语言的表达能力最为重要，它的强弱直接影响到过程模型的适用范围和质量。虽然不同的建模语言因其支持的建模目的不同而在语言的风格与功能上存在差别，但是要构造一个完整且有实用价值的过程模型，不论哪种语言都应该能够描述过程的功能、行为、组织和信息等方面的内容。

· 功能方面

描述软件过程要执行哪些活动，它们的功能是什么，有哪些信息实体与这些活动有关。

· 行为方面

描述什么时候执行这些活动，怎样执行，有哪些行为约束条件。例如进入和退出活动的标准、怎样提供反馈、重复执行的条件与多活动选择执行的决策条件等。

· 组织方面

描述在什么地方，由谁来完成这些过程活动，参与活动和项目的成员的组织结构与成员之间的通信机制等。

· 信息方面

描述由过程活动操作和生成的信息实体，包括数据、文本、中介产品和最终产品、软件对象以及信息实体结构和它们之间的关系等。

目前国际上已出现50多种过程建模语言(Starke 93)，它们在语言的基本成分、语言支持的建模目的、语言采用的形式化方法和语言对软件过程功能、行为、组织和信息的描述能力等方面都存在不同程度的差异。下面我们分别考察几种有代表性的过程建模语言。

① APPL/A

APPL/A(Sutton 90)是过程程序设计语言风格的典型代表。APPL/A 是 L. Osterweil 等人开发用于构造过程程序的一种形式化的建模语言。它是对 Ada 语言的扩充，增加了软件对象之间永久性关系的定义机制、关系操作的触发机制和表示关系状态的谓词机制，并继承了 Ada 语言的基本特征，例如类型系统、模块定义风格和任务通信方式等。

APPL/A 支持对过程的分析、执行控制和复用等建模目的。它通过关系、触发器和谓词等机制可以对过程的功能、行为和对象进行详细的算法描述，可以完全自动地控制过程的执行。但是 APPL/A 不能对过程的人工活动建模，也难以描述过程的动态性和不确定性。此外它也不能描述过程组织方面的信息，使得难以对参与过程的角色进行管理和调度。

② Merlin

Merlin(Peushel 92)是由德国 STZ 公司和德国 University of Dortmund 合作开发的一个以过程为中心的软件开发环境。它的过程建模思想是基于一组良好定义的软件构造块及其属性和关系来构造软件。它支持基于规则的软件过程的执行。Merlin 中的软件过程是用一些规则来描述的，这些规则可以用向前链接或向后链接的方式解释执行。这些规则可以存储在一个面向对象的数据库中成为永久对象被重复使用。Merlin 中使用的建模语言是一种类 Prolog 的语言，这种语言有很大的灵活性和显式声明能力。通过在语言中增加新规则的能力使系统能适应过程的变化需求，从而形成不同的过程模型。过程实施是由环境的推理机制和驱动机制支持的。

Merlin 支持过程复用、过程改善、过程控制与实施等目的，可以描述过程的功能与行为方面的特征。

③ STATEMATE

STATEMATE(Kellner 89)是由美国 Carnegie Mellon 大学 M. I. Kellner 等人使用的一种过程建模语言。它是一种图形式描述语言，由活动图、状态图和模块图组成。活动图用于描述过程的功能特征，主要成分是活动、信息和数据存储方式描述。状态图用于描述过程的行为特征，主要成分是状态和状态之间的变换，其中状态表明活动当前所处的一种工作行为状况，变换则描述了一个状态变换到另一个状态的条件(触发器)。模块图用于描述过程的组织特征，主要成分是模块和信息流，其中模块表示与活动相关的小组或个人，信息流表示了小组或个人之间的信息传递通道。由于 STATEMATE 提供了关于过程的三种视图描述机制，因此大大提高了对过程模型的可理解性。它还提供了一系列对过程进行分析和模拟的设施，以及报告和查询功能等，可对模型进行一致性、完整性和正确性检查。它的缺点是没有提供类型或记录结构等形式的语言机制，从而降低了对产品的分析能力。因此，它主要是一种面向描述的过程建模语

言，不能对过程提供指导与控制。

STATEMATE 以对过程的理解、通信、过程改善与过程管理为建模目的，能较全面地描述过程的功能、行为、组织和信息四方面特征，而且具有良好的可视性。

④ SLANG

过程建模语言 SLANG(Bandelli 94)是 S. Bandelli 等人在 SPADE-1过程驱动环境中为过程建模和执行而开发的。SPADE-1环境由 SLANG 解释器、面向对象数据库及存储设施、同集成化工具子环境相结合的过程解释器接口设施组成。SLANG 基于 Petri 网，提供了特殊的黑色迁移和用户接口位置来管理与外部环境的交互。黑色迁移代表任何外部程序的执行；用户接口位置反映用户或工具产生的相关事件，这些事件被翻译成内部表示与过程实施环境通信。从而使工具和人员之间的交互语义作为过程模型的一部分被描述出来。黑色迁移和用户接口位置是把任务授权给工具和人员的基本机制，也是一种控制外部发生的请求事件的基本机制。

SLANG 支持对过程的理解、通信、人员与工具的交互、过程执行与控制等目的，它可以描述过程的功能、行为等方面特征。

(4) 软件过程与过程模型的度量

目前国际上对软件过程和过程模型的度量与评估还没有形成统一标准，但仍有一些研究方向。

衡量一个软件组织(此处泛指任何一个软件开发单位、开发结构或计算机厂商)的软件过程能力与水平的标准是 CMM (Capability Maturity Model) 框架，即过程成熟度框架(Humphrey 87)。按照 CMM 定义，过程成熟度包括过程计划或定义水平、过程实施水平、过程管理和控制能力和过程改善的潜力等指标的综合。CMM 把过程成熟度分为五级：初始级、可重复级、可定义级、可管理级和可优化级。这五个级别的特点、主要问题与达标标准参见表 10-1。目前国际标准化组织(ISO)处于对过程进行控制和改善的目的，把对软件的度量分为对产品和过程的度量两部分，并且把 CMM 的五个级别转换到 ISO-9003的相应标准中。按 CMM 标准，80年代末美国的调查结果显示，大约87%的软件组织处于第一级，10%—12%处于第二级，只有1%达到第三级，达到第四或第五级的几乎没有。因此，软件组织的过程成熟度与其承担的软件项目的规模和复杂程度存在很大差距，这也是造成当前软件质量和生产率问题的主要原因之一。

虽然没有度量和评估过程模型的标准，但是通过对各种过程模型的分析发现，各种过程模型之间仍然存在着一些共同的质量特征，例如过程模型的功效性、易使用性、准确性、可维护性和可复用性等。研究和改善这些特性对提高过程模型的质量具有重要意义。

表 10-1　过程成熟度模型(CMM)

级　别	名　称	特　点	关键问题	达　标　标　准
1	初始级	过程执行杂乱无序	·项目计划、管理 ·配置管理 ·软件质量保证	过程活动无一定秩序，开发过程的可重复性差
2	可重复级	过程管理工作依赖管理人员的技能	·培训 ·技术复审 ·标准	使项目管理置于严格控制之下，包括严格的项目计划和追踪、子合同管理、需求变化和产品基线控制
3	可定义级	过程可定义、可执行	·过程度量 ·过程分析 ·质量计划	定义一个适合该组织的软件过程，有正规的、文档化的规范，并能根据不同项目的要求裁剪和优化这个软件过程
4	可管理级	过程成为可度量的	·改善技术 ·问题分析 ·防止出错	为定义好的过程建立一套详细的度量机制，为产品和过程设立质量目标，度量软件过程以及产品
5	可优化级	通过反馈来改善过程	·自动化 ·反馈技术	用第四级建立的度量机制，不断地指导过程改善、技术革新和防止出错

第十一章　计算机辅助软件工程 CASE

计算机辅助软件工程这一术语的英文为 Computer-Aided Software Engineering，缩写为 CASE。CASE 使得人们能在计算机的辅助下进行软件开发，为计算机软件开发的工程化、自动化进而智能化打下基础。各国政府和企业部门都很重视 CASE 系统的研制和商品化。在 CASE 工具辅助下进行软件开发，可以提高软件开发效率、改善软件质量。由于 CASE 是软件工程领域的新生事物，国内软件工程课本对它的介绍较少，国外介绍 CASE 时也有许多不同观点。

本章内容分为两大部分，11.1 到 11.3 节是对 CASE 的一般介绍，让大家了解 CASE 这一术语的含义，CASE 系统分类情况，CASE 集成技术，CASE 生命周期如何，较为成熟的 CASE 工作台的具体构成怎样，以及软件工程环境的产生和发展。第二部分，也就是 11.4，介绍了我国自行研制的大型软件开发环境青鸟系统。

11.1　CASE 综述

本节首先说明 CASE 这一术语的含义，然后介绍了 CASE 工具与软件工具的联系，CASE 工具的种类，接下来讨论 CASE 系统的集成技术，最后对 CASE 系统生命周期进行了分析。

11.1.1　什么是 CASE

计算机从 20 世纪 40 年代产生以来，现已广泛应用于工业、农业、军事各个领域和社会生活的各个方面。集成电路的采用，使得计算机硬件得以突飞猛进的发展。硬件的成本极大降低、可靠性大大提高。而计算机软件是智力密集型产品，软件成本十分昂贵，软件质量也因复杂性提高而难以保证。为缓解“软件危机”，60 年代末产生了“软件工程”这门学科。软件工程要求人们采用“工程”的原则、方法和技术来开发、维护和管理软件。

历史上，无论是制造业还是建筑业，当采用强有力的工具辅助人工劳动时，生产率就得到了极大的提高。例如在建筑新房时，一个人使用推土机后，一天中可移动的土超过 50 个人用手工工具移走的土。相似的，当工程设计师使用 CAD 系统时，他们的生产率就大大提高了。CAD 系统取代了繁锁的绘图工作，还能检查出设计的错误和遗漏处。

因而，可以想见，软件工程师使用辅助软件设计的自动化工具后，软件生产率肯定能得到提高。20 世纪 80 年代早期，就涌现出了许多支持软件开发的软件系统。术语计算机辅助软件工程(CASE)现在已被作为软件工程界的这种自动化支持的代名词而普遍被接受。

具体地说，CASE 是一组工具和方法的集合，可以辅助软件开发生命周期各阶段进行软件开发。CASE 计算机辅助软件工程，广义地说，是辅助软件开发的任何计算机技术，它有两个主要的含义：① 在软件开发和/或维护过程中提供计算机辅助支持；② 在软件开发和/或维护中引入工程化方法。

CASE 既涉及学术研究领域，又涉及产业领域。从学术研究角度讲，CASE 是多年来在软

件开发管理、软件开发方法、软件开发环境和软件工具等方面研究和发展的产物。CASE 把软件开发技术、软件工具和软件开发方法集成到一个统一而一致的框架中，并且吸收了 CAD(计算机辅助设计)、软件工程、操作系统、数据库、网络和许多其它计算机领域的原理和技术。因而，CASE 领域是一个应用、集成和综合的领域。从产业角度讲，CASE 是种类繁多的软件开发和系统集成的产品及软件工具的集合。其中，软件工具不是对任何软件开发方法的取代，而是对方法的辅助，它旨在提高软件开发的效率，增强软件产品的质量。

在 90 年代中期，有许多 CASE 生产商，提供了成百上千种 CASE 产品。在 20 世纪 80 年代后期和 20 世纪 90 年代，这些产品的市场发展非常迅猛，但现在似乎已进入一个增长较慢的阶段，这归因于经济环境。然而，第一代 CASE 产品并没有带来其供应商所预言的极大提高生产率的现象。

究其原因在于：

① Broods 于 1987 年已尖锐地指明，大型系统开发的根本问题是被开发的产品复杂性和开发过程复杂性的问题。他还指出，CASE 技术能提供某些支持来控制或降低复杂性，但不能最终解决复杂性这一根本问题。

② 当前 CASE 产品代表“自动化岛”，虽然可以或多或少地支持各种过程的活动。但这些产品间的集成是有限的，从而限制了该技术的应用。

③ 采用 CASE 技术的人时而低估培训费用，而这对有效地采用 CASE 是很关键的。

尽管使用 CASE 产品时，要考虑它的性能价格比，但 CASE 的确可以降低软件成本，缩短开发时间，提高软件质量。CASE 的支持使得某些软件过程的再工程成为可能。因此，CASE 已对软件工程的发展作出了贡献，并将继续作出贡献。

11.1.2 CASE 分类

1. CASE 技术种类

CASE 系统所涉及到的技术有很多。从软件开发和管理角度，CASE 技术有两类：一类是支持软件开发过程本身的技术，如支持规约、设计、实现、测试等等。采用这类技术的 CASE 系统研制时间较长，已有许多产品上市；另一类是支持软件开发过程管理的技术，如支持过程建模、过程管理等等。这类技术不很成熟，采用这类技术的 CASE 系统会调用采用了前一类技术的 CASE 系统。

从 CASE 系统产生方式来看，还有一种特殊的 CASE 技术，即元-CASE 技术。元-CASE 技术是生成 CASE 系统的生成器所采用的技术。该生成器可用来创建支持软件开发过程活动及过程管理的 CASE 系统。此类 CASE 技术尚处于探索阶段。

2. CASE 工具

在 CASE 术语尚未广泛使用之前，人们就经常使用软件工具一词。20 世纪 70 年代末 80 年代初，软件工具的含义极为广泛。凡是用于辅助或支持计算机软件的开发、运行、维护、模拟、移植或管理而研制的程序系统都称为软件工具。从这种广泛的含义上讲，软件工具可谓包罗万象，它既包括比较成熟的系统软件，如操作系统、编译程序、解释程序和汇编程序等，又包括支持软件开发各个方面的程序。

软件工具的发展有以下特点：

① 软件工具由单个工具向多个工具集成化方向发展。如将编辑、编译、运行结合在一起构

成集成工具。注重工具间的平滑过渡和互操作性。如微软公司的 Office 工具。

② 重视用户界面的设计。交互式图形技术及高分辨率图形终端的发展，为友好方便的用户图形提供了物资基础。多窗口管理、鼠标器使用、图形资源的表示等技术，极大地改善了用户界面的质量，改善了软件的感观。

③ 不断地采用新理论和新技术。如许多软件工具的研制中采用了数据库技术、交互图形技术、网络技术、人工智能技术和形式化技术等。

④ 软件工具的商品化推动了软件产业的发展，而软件产业的发展，又增加了对软件工具的需求，促进了软件工具的商品化进程。

随着计算机软件的发展，这种含义上的软件工具越来越多，甚至像数据库管理系统也可称为软件工具。因而，人们开始使用软件工具的窄一些的含义，即软件工具是用于辅助计算机软件的开发、运行、维护和管理等活动的一类软件。随着 CASE 的出现，人们也经常使用 CASE 工具这一术语。人们一般不加区别地使用软件工具和 CASE 工具这两个词。但它们还是有细微的区别的。

CASE 工具有两层含义。狭义上来讲，它是一种特殊的软件工具，用于辅助开发、测试、分析和/或维护另一个计算机程序和/或其文档。在这种意义上，CASE 工具是不包括系统软件的软件工具。广义上讲，它是除操作系统之外的所有软件工具的总称。本书谈及 CASE 工具时，一般是指其广义上的含义。

CASE 工具的粒度可大可小，小到可以是如一个单独的测试程序，大到可以是一个集成有多种工具的软件环境。像源程序分析和测试工具，软件管理、控制和维护工具，需求/设计规约工具，程序构造和生成工具，软件模型和模拟工具，以及软件/程序设计支撑环境等都是 CASE 工具。

3. CASE 工具的分类

CASE 领域发展很快，出现了各种各样的 CASE 工具。如何对 CASE 工具进行分类呢？有必要确定一个分类模式来评估和比较不同的工具。分类既有助于人们在一个统一的概念基础上来理解 CASE 工具，也有助于向 CASE 工具潜在的用户解释这一领域。

表 11-1 列出了许多不同类型 CASE 工具，给出每类工具的 CASE 系统实例。

表 11-1 CASE 工具的功能分类

工具类型	实例
管理工具	PERT 工具，评估工具
编辑工具	正文编辑器，图形编辑器，字处理器
配置管理工具	版本管理系统，改变管理系统
原型工具	甚高级语言，用户界面生成器
方法支持工具	设计编辑器，数据字典，代码生成器
语言处理工具	编译器，翻译器
程序分析工具	交叉引用生成器，静态分析器，动态分析器
测试工具	测试数据生成器，文件比较器
调试工具	交互式调试系统
文档工具	页面布局程序，图像编辑器
再工程工具	交叉引用系统，程序再构造系统

对 CASE 工具分类的标准可以分为：

(1) 功能

功能是对软件进行分类的最常用的标准。表 11-1 就是基于功能分类的。

(2) 支持的过程

根据支持的过程，工具可分为设计工具、编程工具、维护工具等等。

(3) 支持的范围

根据支持范围，可分为窄支持、较宽支持和一般支持工具。窄支持指支持过程中特定的任务，如创建一个实体关系图，编译一个程序等。较宽支持是指支持特定过程阶段，例如设计阶段。一般支持是指支持覆盖软件过程的全部阶段或大多数阶段。

图 11.1 是一个基于活动的分类，显示了许多不同类 CASE 工具所支持的过程阶段。针对规划和评估、编辑、文档准备和配置管理的工具可用于软件过程的全阶段。

CASE 工具能支持的活动有需求定义、形式化规约、面向功能设计、数据建模、面向对象设计、编程、测试、维护和管理等。就目前而言，CASE 工具对设计和实现活动能提供很好的支持，但对需求和维护的支持就要弱很多。支持最好的活动是面向功能的设计、数据建模、面向对象设计和编程；支持较好的活动是需求定义、测试和管理；支持最差的是形式化规约和维护。

	规约	设计	实现	验证和校验
测试数据生成工具				✓
建模和模拟工具	✓			✓
程序转换工具			✓	
交互式调试系统			✓	✓
程序分析工具			✓	✓
语言处理工具			✓	✓
方法支持工具	✓	✓		
用户界面管理工具		✓	✓	
数据字典工具	✓	✓		
图编辑工具	✓	✓		
原型工具	✓		✓	
配置管理工具	✓	✓	✓	✓
文档准备工具	✓	✓	✓	✓
正文编辑工具	✓	✓	✓	✓
规划和评估工具	✓	✓	✓	✓

图 11.1 CASE 工具基于活动的分类

系统建模、设计和编程工具是质量最好的 CASE 工具，这反映对这些活动已有相当的了解。CASE 还不能很好地支持需求定义、形式化规约和维护等活动。当 CASE 工具基于某一方法时它是最成功的。当前还没有针对这些活动的有效方法。

过去十多年间，CASE 领域发展突飞猛进，新的 CASE 产品不断涌现。人们觉得用“CASE 工具”一词不足以描述新的 CASE 产品，因而使用了工具、工具箱、工作台和软件工程环境等术语来描述不同的 CASE 产品。

CASE 产品可笼统地称为 CASE 工具。但为了讲述的便利，在本章其后几节采用 Fuggetta 的分类方法介绍 CASE 系统。

1993 年，Fuggetta 根据CASE 系统对软件过程的支持范围，提出CASE 系统可分为三类：

① 支持单个过程任务的工具，例如检查一个设计的一致性，编译一个程序，比较测试结果等等。工具可能是通用的，独立的(如一个字处理器)，或者也可能归组到工作台。

② 工作台支持某一过程阶段或某些活动，例如规约，设计等等。它们一般以或多或少的集成度组成工具集。

③ 环境支持软件过程所有活动或至少大体部分。它们一般包括几个不同的工作台，将这些工作台以某种方式集成起来。

表 11.2 图示说明了该分类，并给出了这些不同类别的 CASE 所支持的一些例子。当然，许多类型的工具和工作台都未包含在该图中。

工作台一般支持某种方法，该方法包含一个过程模型和一组规则/指南，以应用来开发软件。工作台因而应提供一些指导说明如何应用该方法，以及何时使用工作台中的工具。

本书将环境分为集成化环境和以过程为中心的环境。集成化环境对数据、控制、表示集成提供基本支持。以过程为中心的环境中包括软件过程知识和一个过程机(过程引擎)。该过程机用过程模型来指导软件开发人员使用什么工具或工作台，以及何时使用它们。

实际上，这些不同类别之间并无明显界限。作为单独产品出售的工具，也许能嵌入到别的工具中以支持不同的活动。例如，字处理器中嵌入的图表编辑器，图表编辑器既能作为一个单独的工具，又能作为字处理器的一个子系统。随着 CASE 工作台的成熟，随着 CASE 工作台在功能更强大硬件上运行，CASE 工作台对编辑、编译和测试的支持不断增强，从而它们在功能上接近一个软件工程环境。因此，这一分类只是一个粗略的分类，以方便大家对 CASE 系统的理解。

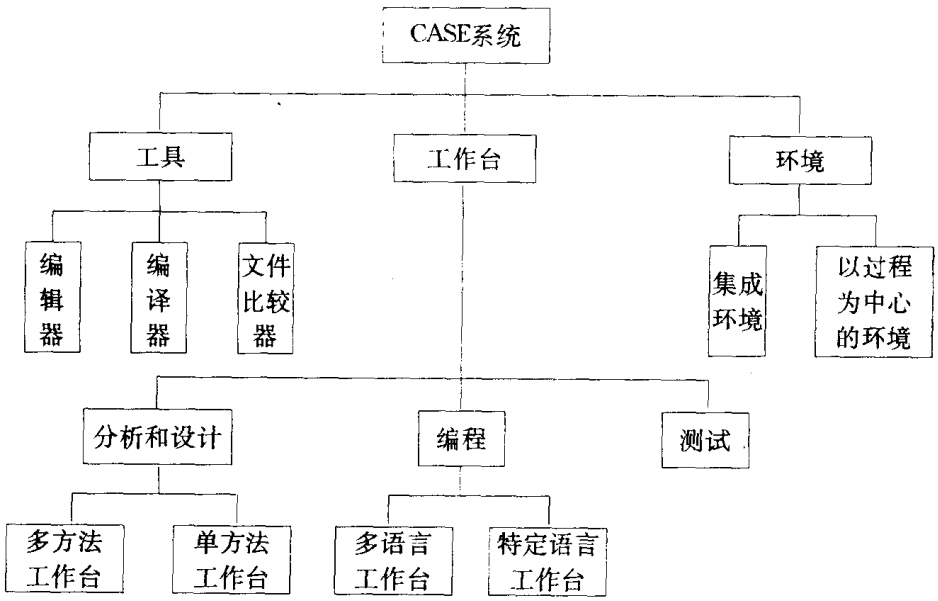


图11.2　工具、工作台和环境

11.1.3　集成化 CASE

通常，单独的 CASE 工具是很有用，而且性能价格比很高。但是，当 CASE 工具以一种集

成的方式工作时可获得更多收益。集成方式的优势在于组装特定工具以提供对过程活动更广泛的支持。一个有效的集成框架使得演进成为可能，当加入新的系统时，不会影响原有系统。如果CASE 系统用户界面集成了，就可能减少用户学习时间、降低用户错误率。

一个集成系统中各子系统能彼此利用对方提供的功能，如一个设计工作台和一个文档工作台集成后，设计工具自动生成的文档能使用文档工作台进行排版并加入到文档工作台所产生的系统文档中。

1990 年 Wasserman 讨论软件工程环境的集成时，提出一个五级模型。这一模型也适用于工作台。

①平台集成：工具运行在相同的硬件/操作系统平台上。

② 数据集成：工具使用共享数据模型来操作。

③ 表示集成：工具提供相同的用户界面。

④ 控制集成：工具激活后能控制其它工具的操作。

⑤ 过程集成：工具在一个过程模型和“过程机”的指导下使用。

从用户的角度看，集成意指 CASE 工具显示了某种一致的度量。对不同的集成系统，它们的一致性差别很大。松散耦合系统也许只提供了有效的数据互换。相反，紧耦合集成系统的工具在单一、共享的软件表示上操作，使用一个一致的用户界面，从一个工具能平滑过渡到另一个工具(即无缝转移)。

1. 平台集成

平台集成是指工具或工作台在相同的平台上运行，其中“平台”或是一个单一的计算机或操作系统或是一个网络系统。目前，大多数 CASE 工具运行于 UNIX 系统，或 PC 上的 Microsoft Windows 之上。

当一个组织机构使用异构网络，网络中不同的计算机运行不同的操作系统时，要实现平台集成很困难。即使机器全是从同一个供应商处购买，平台集成仍是一个问题。新机器也许带有新的操作系统版本，而与新机器安装在同一个网络中的旧机器也许运行的是该操作系统的旧一些的版本。有些情况下，已有的 CASE 系统不能立即在新操作系统上运行。新购买的 CASE 系统也许不能和该操作系统的旧版本一起工作。

2. 数据集成

数据集成指不同软件工程环境能相互交换数据。因而，一个工具的结果能作为另一个工具的输入。

有许多不同级别的数据集成：

(1) 共享文件

所有工具识别一个单一文件格式。最通用的可共享文件是字符流文件。

(2) 共享数据结构

工具使用的共享数据结构通常包括有编程和设计信息。事前，所有的工具要认可该数据结构的细节，并把该结构细节“硬件化”进工具中。

(3) 共享仓库

工具围绕一个对象管理系统来集成，该 OMS 包括一个公有的、共享数据模型来描述能被工具操纵的数据实体和关系。这一模型可为所有工具使用，但不是工具的内在组成部分。

最简单的数据集成形式是基于一个共享文件集的集成，UNIX 系统就是这样。UNIX 有一

个简单的文件模型，即非结构化字符流。任何工具都能把信息写入文件中，也能读其它工具生成的文件。UNIX 文件系统把 I/O 设备作为特殊文件来处理。它还提供管道。当进程间通过一个管道联系时，无须中间文件创建，字符流从一个进程直接流向另一个。

文件是一个用于信息交换的物理简单方法。然而，应用程序如果利用文件来访问另一个工具产生的信息时，就必须知道一个文件的逻辑结构。这个逻辑结构嵌入在写这个文件的程序中。或者所有工具依从哪个格式，该格式成为一种标准，或者使用信息的工具必须知道该信息是哪个工具创建的。

一个基于文件的集成策略导致点对点的集成方法。每对工具必须认同文件互换格式，或者由一个过滤程序来完成该共享文件格式从一个表示到另一个表示的转换。参见图 11.3。

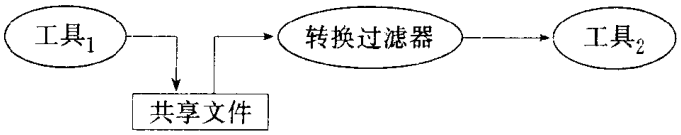


图11.3　基于共享文件的点到点的工具集成

原则上，共享文件集成需要为每一对要集成的工具编写一个转换过滤器。因为不同工具使用的格式可能差别相当大，写这些过滤器的开销也很高。1993 年，Rader 等对 CASE 工具集成作了研究，报告说，某些情况下，需要 1—2 年的努力来实现点到点集成的转换过滤器。

另一种方案是，认可文件格式约定，编写工具时参考这些约定。这是 UNIX 系统早期版本使用的方法。UNIX 系统是最早的有效的程序设计环境之一。然而，新的工具也许发现这些被认同的约定限制太多，因而忽视这些约定，使用自己的格式，这些工具很难和系统中已有工具兼容。

1991 年，许多 CASE 工具销售商提供了一个 CASE 数据交换格式，这可作为 CASE 系统的一个标准数据交换格式。在写本书时，已经制定了标准提案，但似乎对可供使用的 CASE 系统没有什么重大影响。这个标准的未来，以及 CASE 系统销售商支持的程度都不清楚。

采用共享语法和语义信息的数据集成的另一个可选的方法是基于共享数据结构的。这些数据结构也许代表了诸如数据流图、实体关系图或程序设计语言等术语。有许多编程语言转换和支持工具参与这种集成方式。这些工具依赖于对程序的语法语义分析。生成的代码与符号表和语法树相连，因此程序执行信息能用高级语言术语来表示。图 11.4 图示了一个程序设计语言工作台中这种形式的数据集成。

围绕共享数据结构的集成隐藏了工具间的差别，呈现给用户的是一个一致的程序开发系统。然而，很难将工具移到这个环境中，原因在于数据集成表示过于复杂。任何新工具必须知道该共享结构的细节。这通常是编译器开发商的专利。因此，基于这种形式集成的工作台通常是单独的系统，和其它 CASE 工具集成的唯一可能形式是通过程序设计语言源代码。

围绕一个仓库或对象管理系统的集成是适应性最强的一种数据集成形式。数据管理系统是一个数据库系统，包括输入实体到系统，将属性与这些实体相联，在实体间建立分类关系等设施。

这种集成方式的关键是要设计一个公共的数据库模式。在该数据库模式中定义了实体类型和实体间的关系。工具根据这一模式来读写数据。如果一个工具希望使用另一个工具产生

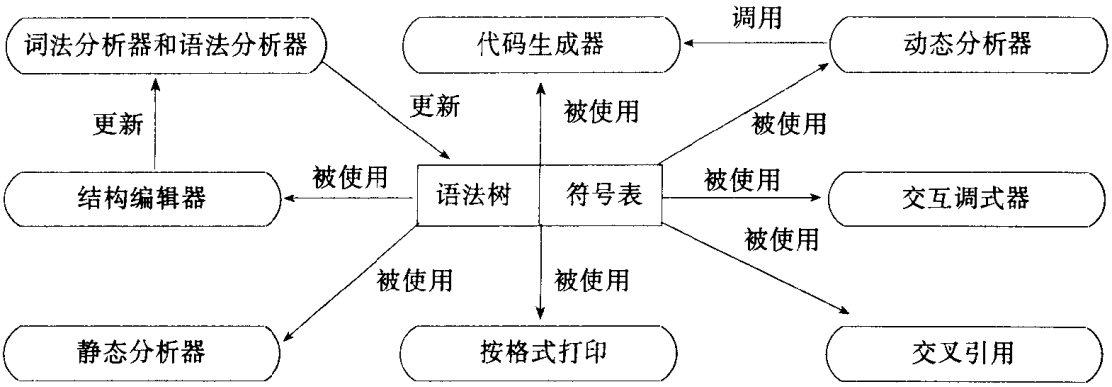


图11.4　通过共享数据结构的集成

数据，那么将用到该模式来发现那个工具所生成的数据结构(图 11.5)。

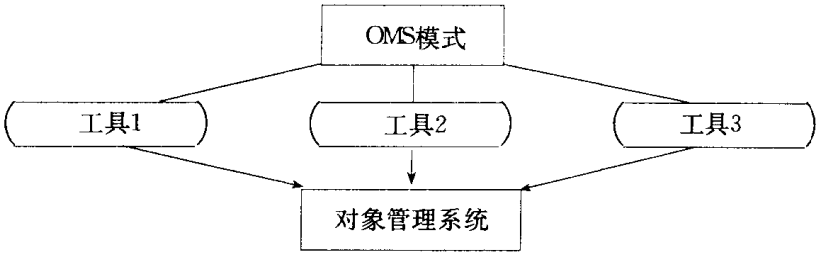


图11.5　通过OMS集成

这种数据集成的方法主要有两个缺点：

① 工具要独立编码以利用OMS。已有工具若要利用OMS 就得付出相当的努力。

② 除工具或工作台外，CASE 用户还必须购买 OMS。这又增加了开销和风险，许多公司都不愿承担。

自从 80 年代初以来，一直在进行有关 OMS 的研究，已经定义和实现了像 PCTE 这样的环境框架。然而，CASE 工具销售商已表示不愿采用任何框架标准。他们大多喜欢用自己的专有方法。这就意味着，基于这种集成形式的 CASE 系统很有限，并且依靠一些特定的组织机构。

3. 表示集成

表示集成或用户界面集成意指一个系统中的工具使用共同的风格，以及采用共同的用户交互标准集。工具有一个相似的外观。当引入一个新工具时，用户对其中一些用户界面已很熟悉，这样就减轻了用户的学习负担。

有三种不同级别的表示集成：

(1) 窗口系统集成

这一级集成的工具使用相同的基本窗口系统，窗口有共同的外观，操作窗口的命令也很相似，如每个窗口都有窗口移动、改变大小、图标化等命令。

(2) 命令集成

这一级集成的工具对相似的功能使用同样格式的命令。如果使用一个文本界面，所有命令都使用相同的命令和参数的语法格式。如果使用一个用菜单和图标的图形界面，相似的命令就

会有相同的名字。在每个应用程序中菜单项定位于相同位置。在所有的系统中，对按钮、菜单等使用相同的表示(图标)。

(3) 交互集成

这种集成是针对那些带有一个直接操纵界面的系统，通过该界面用户可以直接与一个实体的图形或文本视口进行交互。交互集成意指在所有子系统中提供相同的直接操纵操作，如选择、删除等等操作。因而，如果通过双点按选择正文，那可能在一个图表中也以相同方式来选择实体。支持交互集成的系统例子有字处理系统和图形编辑系统。

命令集成意指以相同方式来支持应用程序和环境的控制功能。例如，所有应用程序需要一些机制来允许用户终止其执行。而所有的应用程序可能有相同的"quit"按钮。如果工具是文本命令，那么命令应有相似或相同格式的参数名。

如果实现都按照一组指南来定义抽象用户界面操作的表示方式，就能实现命令集成。这些表示可能包括从可选集中作一个选择，拴住一个开关，显示数字或字符信息等。这样的指南的一个较早的例子是 Sommerville 等在 1989 年介绍的 ECLIPSE 环境中给出的有关定义。根据菜单、按钮、显示框、开关的"复合"等来定义用户界面。

大多数软件工具都用图形或正文来描绘它们所操纵的对象。不管这些对象是什么，交互集成要求与这些图形或文本对象交互时所使用的机制是一致和统一的。例如，如果正文一般经光标控制键横过正文来选择正文段，那么需要选择正文的所有工具都要使用同样方法。

为交互集成提供指南特别困难。这是因为可能交互的数目、文本和图形对象表示的可能的范围都难以确定。为非结构化文本和未分类的图形对象定义其可能的交互行为相当容易，但当文本或图形代表一个结构化实体，其操作是通过屏幕外观来进行时就困难多了。

在开放系统中，很难实现基于窗口系统级的用户界面集成。在这些环境或工作台中，工具是由不同的开发人员在不同的时期开发的。随着系统演化和新工具的引入，很难维护用户界面的一致性。尽管可以提供包罗万象的用户界面集成指南，但指南设计人员不能预知环境的每一个可能的使用。新的设施，例如声音的使用，也许不为指南所考虑，因而就会引入各自的有关声音的用户界面约定。这一过程中不可避免地就会失掉统一性。

1993 年，UNIX 工作站窗口系统集成的事实标准是 X 窗口 Motif 工具箱。实际上所有基于 UNIX 的 CASE 工具都认可这一标准。然而，它只解决了窗口级的集成。来自不同销售商的 UNIX 工具和工作台很少很好地在命令和交互级上集成。这些 CASE 系统也许提供命令剪裁机制来提高表示集成。但是，通常是不支持用户细节操作的交互集成的。

在 PC 平台上，事实上的标准是 Microsoft Windows。这比 X/Motif 好得多，因为它支持三个级别的表示集成。Windows 除具有通常窗口系统所应具有的功能外，还提供工具箱来支持其所制定的菜单构造指南，还支持特定形式交互。实现一个工具界面最好的方法是遵照这些指南，这样，运行了 Windows 的 CASE 工具都有一致的外观。

CASE 工作台是封闭系统时，通常都有良好的用户界面集成。由于不能由其它供应商引入新工具，因而工作台的工具很容易具有用户界面的一致性。工作台的不同工具能遵从一些约定和标准。用户能从一个工具无缝地移至另一工具。

4. 控制集成

控制集成支持工作台或环境中一个工具对系统中其它工具的访问。除了能启动和停止其它工具外，一个工具能调用系统中另一工具所提供的服务，这些服务可通过一个程序接口访问

到。例如，一个综合工具箱中，一个结构化编辑器可以调用一个语法分析器来检查所输入的程序片断的语法。

一些工作台针对控制集成开发了特定机制。然而，一些销售商已采用基于消息传递的更为通用的方法。这已在 Sun 的 ToolTalk 等系统中实现。1995 年 Brown 描述了这种工具集成的方法并比较了消息传递系统的几种实现机制。

在消息传递方法中，CASE 系统通过传递消息来彼此互换信息。这些消息能提供状态信息，通知其它工具正在进行什么，或者请求特别的服务，该服务是由一个 CASE 系统提供的。一个消息服务器管理 CASE 系统间的通信。

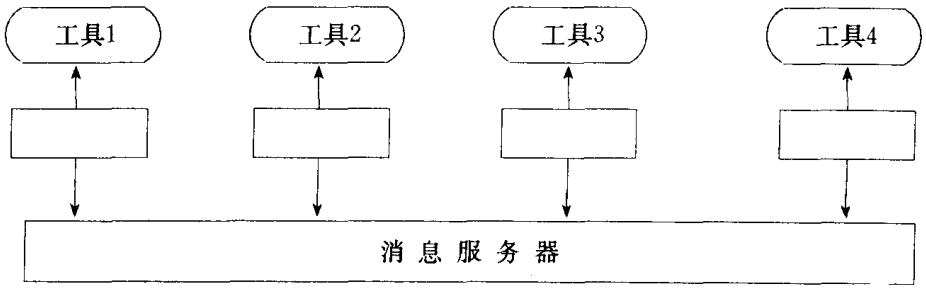


图11.6　通过消息传递的控制集成

图 11.6 说明了这个通用模型。参与集成的每个工具提供一个控制接口，通过控制接口可以访问该工具。消息服务器了解所有可访问工具的接口信息和这些工具的定位。它负责网络传输的编码和解码。

当一个工具需要和另一个工具通信时，它用已知的格式构造一条消息，写上地址，把这个消息发送到消息服务器。工具不必知道被调用的工具在那儿，它只简单地告诉消息服务器，服务器再将该消息传给被调用的工具。因而，该模型支持分布式 CASE 系统，即系统的不同部件在不同的计算机上运行。

逻辑上，工具的广播消息被对之感兴趣的其它工具接收。实际上，这是一个低效的方法。消息服务器知道每个 CASE 系统能处理的消息，因而只传递合适的消息。为说明这一方法，假设 CASE 系统包括一个设计编辑器，一个代码生成器和一个编译器。这些都是分别实现的。作为一个编辑会话的结果，下列动作可能发生：

① 设计编辑器发给代码生成器一条消息，请求处理设计，并生成代码。

② 生成代码后，代码生成器发送一个能被设计编辑器和编译器两者截获的消息。该设计编辑器将生成的代码文件与设计相连，编译器编译所生成的代码。

③ 在代码已经生成后，编译器发送一个消息，被设计编辑器截获，它告之用户编译已完成。

控制集成也需要某种级别的数据集成，以便能互换工具操作的参数。一个操作期间要互换的数据格式通常是用一个接口定义语言(IDL)来编写的。该 IDL 引入了每个工具能使用的一组标准类型。每个工具必须将要互换的数据转换成这些类型，这样，它能被接收工具处理。

但是，这种方法只适合于互换相当短的消息。大块的数据交换还必须通过文件或对象来组织。因此，系统间传递的消息一般包括对存有共享数据的文件的引用。

5. 过程集成

过程集成意指 CASE 系统嵌入了关于过程活动、阶段、约束和支持这些活动所需的工具

的知识。CASE 系统辅助用户调用相应工具完成有关活动，并检查活动完成后的结果(图11.7)。

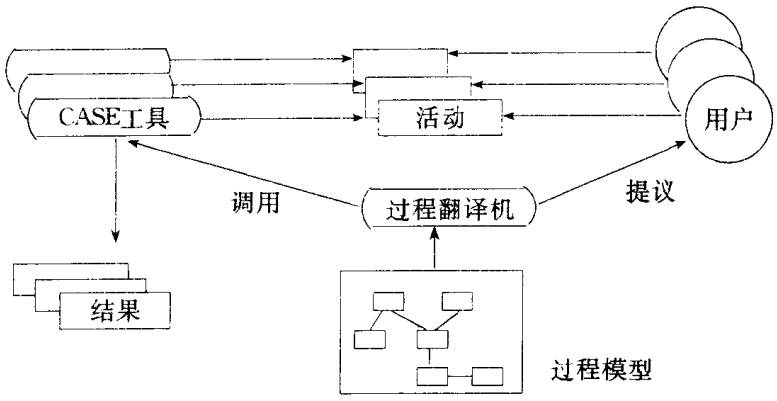


图11.7 过程集成

过程集成需要CASE 系统维护软件过程模型，并实例化和实施该软件过程模型。其间，要说明有哪些活动，活动之间的过渡情况如何，完成活动要哪些工具支持，整个开发采用了哪些策略等等。所有这些都在软件过程模型里表示出来，由一个过程翻译器或“引擎”实施该模型，在软件过程驱动下进行软件开发。

目前认为过程实施是不可预知的，但应给过程工程师提供指南以处理过程活动。也认识到不是所有的活动都能建模或给予支持。过程支持系统有一个“可选项”允许部分处置由开发人员来自行决定。

许多活动是并发活动，这点必须反映在过程模型中，活动是相互依赖的，因而过程模型应是动态的。并且随着所获取的与过程活动有关的信息越来越多时，过程模型应可改变。

CASE 技术对过程集成的支持依赖于过程模型的设计。谈到过程建模，要明确以下问题：

① 在没有过程支持的环境中开发软件时，也常谈到过程模型，这种情况下谈及的过程模型是类属模型。它们靠人类翻译，以在任何特定环境中实例化该模型。在这些过程模型中未定义活动及其实现的细节。

② 决没有一条简单的捷径来组织软件开发，系统开发时也没有一条捷径让项目管理者和开发人员随意改变过程。如果发生未预料的事情，人们可以很快地在活动间切换(例如，打印机不能打印)。但是在一个模型中嵌入这种随意性很难。

③ 过程模型说明软件过程的产品和开发人员之间的通信。对诸如发货处理这种结构良好的任务，可以形式化描述其通信情况。但是，要把软件开发的组织形式、如何解决具体突发问题的处理模式说清楚很困难，更不必说形式化了。

过程集成中存在的较大的阻力可能来自开发人员和经理，他们可能已经习惯用非形式的、文档化的方法进行软件开发和管理，不习惯用环境中提供的严格策略。这样，他们可能不愿意使用支持过程集成的CASE 系统。

现在，已有许多过程建模工作台，例如 Process Weaver(Fernstrom，1993)能定义和实施过程模型。这些过程模型能用以上讨论的一个控制集成框架与其它 CASE 工具连在一起。已经建立了许多的以过程为中心的环境。但还没有发展成商用产品。在 Rader 等 1993 年的研究中，尚未发现对这种集成的商品需求。

11.1.4 CASE 生命周期

引入和使用CASE 技术需要仔细地筹划。CASE 工具很昂贵，必须维护这些工具直至使用这些工具生产的所有软件都过时为止。如果对 CASE 需要量不是很大，使用这种技术可能很难获得收益。如果作出了错误决定，当 CASE 系统投入使用时，软件成本就会增加而不是降低。

一个组织中的CASE 系统遵循从初始需求到完全废弃这一生命周期(图 11.8)。

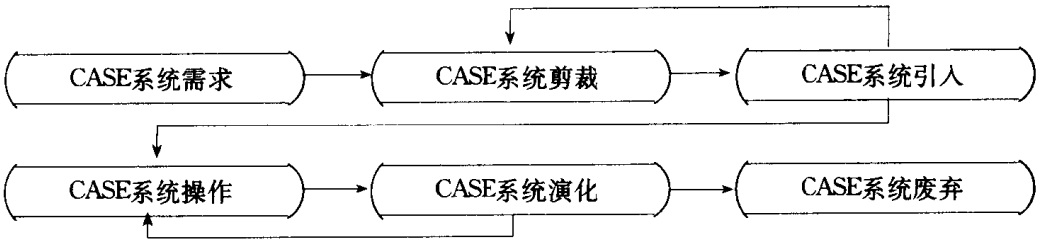


图11.8 CASE系统生命周期

CASE 生命周期各步骤如下：

① 需求：根据要开发的软件类型选择一个合适的 CASE 系统。

② 剪裁：调整一个 CASE 系统，使之适应一特定组织机构或一类项目。

③ 引入：试用该 CASE 系统。在这期间，要培训使用这一系统的开发人员。

④ 操作：每天都使用 CASE 系统进行软件开发。

⑤ 演化：演化事实上不是一个单独的步骤，而是一个在 CASE 系统生命期中的一个持续的活动。要修改硬件或软件，调整系统适应新需求。

⑥ 废弃：CASE 系统在这一阶段不再起作用，必须保证使用该系统开发的软件仍被所在组织机构所支持。

显然，从引入阶段到剪裁阶段有反馈。在演化和操作之间也有反馈。

引入 CASE 时，必须有意识地进行管理，让开发人员认识到 CASE 系统的优势所在。CASE 系统的开销收益是长期的，而不是短期的，不可能立即节省开销。

1. CASE 需求

现在能提供成百上千种 CASE 系统。最主要的是分析和设计工作台。这些工作台大多数都很相似，至少在表面上很相似。当然，也有许多其它种类的 CASE 系统，从专用工具到工作台到环境都有。

当需要一个可在整个公司范围使用的 CASE 系统时，必须考虑以下因素：

(1) 已有的公司标准和方法

CASE 系统应支持已有的实践，而不是引入新的标准和方法，还有可能为开发人员拒绝。

(2) 已有的计算机和将来购买的计算机

购买一个能在已有计算机上运行的系统是明智的。但是，如果当前硬件负载过大，已没有空间来容纳 CASE 系统，就需要投资购买新机器。环境的生命期很长，因而，购买一个不能在工业标准的计算机上运行的系统是不明智的。

(3) 要开发的应用程序类型

很显然，所选择的 CASE 系统应能为购买它的组织机构提供方便，具体而言，能在该组织

机构开发所需要的应用程序时提供便利。如果组织机构中，大多与开发科学计算 FORTRAN 程序有关，那么就不应购买一个商用开发的 CASE 系统。

(4) 安全性

CASE 系统应提供合适的安全性特性，使得安全经理能说明环境中的访问权限。大多数 CASE 工作台没有特别考虑安全性。

其它主要是 CASE 系统的费用。1992 年 Huff 估计一个可靠的 CASE 系统购买费用是近于每人 $18000，5 年的支持费用每人 $40000。这个数字显然在不同组织机构间变化很大。这也与 CASE 类型和相关支持平台有关。但是，这至少说明对一个组织机构来讲，CASE 系统会有很大的投资，因而对 CASE 系统的需求决定是很重要的。

2. CASE 系统剪裁

CASE 系统是通用软件系统，将 CASE 系统安装在一个组织机构中时必须对它进行改写，以适应特定的要求，并使之在该组织中发挥作用。针对特定组织和应用领域定制一个 CASE 系统需要的活动包括：

(1) 安装

系统必须在该组织机构的硬件配置下安装和测试。这可能涉及依赖参数而改变系统。要编写脚本以确保系统能正确安装、并支持系统的使用。

(2) 过程模型定义

即使 CASE 系统不是过程驱动的，也需要某些过程模型来说明需要进行哪些系统剪裁。过程模型允许 CASE 管理者了解在哪儿使用工具、必须构造怎样的接口来与其它工具相连。

(3) 工具集成

新的 CASE 系统也许不得不与其它已安装的 CASE 系统集成。如果系统是通过一个共享数据管理系统集成的，那么必须定义和校验数据模式。这涉及到要确认在一个组织的软件开发过程中需要的所有实体和关系。正确获取模式是成功集成的关键。因而，这一步可能要花上几个月的时间。如果系统通过文件集成，那么必须确认共享文件格式，在某些情况下，还要编写转换过滤程序。

(4) 文档

作为剪裁过程的一部分，CASE 系统特定的初始过程必须文档化。

剪裁所需要的时间和所涉及的开销是不能低估的。除非是一个非常简单的系统，否则剪裁一个 CASE 系统总要花上几个月时间，在一个有效系统可提供之前，还可能花一年多时间呢。

3. CASE 系统引入和操作

将一个 CASE 系统引入一个组织，不可避免地会改变工作方式。直到对 CASE 系统使用一段时间后，才可能从中获益。可能有预知的困难发生，也可能有不可预知的问题产生，必须提供处理这些问题所需的资源。

当引入一个 CASE 系统时，可能发生如下问题：

(1) 用户阻力

很少有例外，人们天生保守，并且除非有明显好处，否则会拒绝新的发展。CASE 对管理有帮助，因它一般对软件过程提供了更多控制。个体软件开发者所获得的好处就不明显了。有争议说 CASE 系统规定和限制了个体工程师的创造力。

(2) 缺少培训

一些软件开发人员也许觉得新的开发系统本身很难，他们对系统的某些设施也不了解。这对那些缺乏软件工程上正规训练的员工很可能是一个问题，因为软件开发方式上改变的步伐太快。他们也许已经习惯了使用其它一些系统，不愿花时间去学习一个新系统。

(3) 管理阻力

一些经理不愿将CASE技术引入到一个已知的开发过程。经理们负责控制开支，但引入CASE的费用开支很难事先量化。使用一个难以预测的支持系统增加了开发的风险。

这些反对理由在某些场合确有其道理。然而，进行系统设计时，CASE工具可以辅助处理锁碎的事务，这些工作已成为软件开发的一部分(例如设计图表，找到相关代码和文档等)。CASE系统可以让个体开发人员有更多时间来做创造性的工作。CASE不会降低软件开发人员的技能。

对所有要使用CASE系统的软件开发人员进行适当培训，可以解决第二个异议。这是一笔开支，并且经理必须提供培训的开销。CASE系统应与已有支持系统一起介绍给公众，就样，就可以向其它项目的员工演示CASE系统的好处了。

最后的异议，是管理阻力，很难克服。采用新的软件过程技术就像跳入了一个未知的深渊。在管理部门意识到它之前，必须有明显证据表明该方法的优势。

指派技术是很少成功的。关于CASE系统的购买，与项目经理管理经验的关系也很大，另外，CASE技术也应一步步介绍。新项目开始使用CASE支持时，应认真考虑花销和项目开发人员和经理的经历。在有人使用过CASE之后，要介绍这些人的使用情况，让其它经理意识到CASE系统的作用。

4. CASE系统演化

如同其它软件系统，CASE系统要保证有用，就不能是静止不变的。当出现新的需求，当可以使用新的硬件和软件平台时，CASE系统必须演化。也许要增加新的CASE工具，改变输入输出格式，使用新方法进行软件开发等。

CASE系统演化的问题之一就是新旧版本可能不兼容。不可能用系统的新版本来维护所有软件，因而，在任何时刻就有可能使用CASE系统的几个版本。如果不兼容性是由于硬件改变了，那这意味着也必须维护旧的硬件以支持软件系统。如果不兼容性与操作系统有关，那么就需要有异构网络，支持不同的机器运行不同版本的操作系统。

5. CASE系统废弃

在CASE系统生命周期的某些阶段，也可能决定用新的CASE系统替换它，因为CASE系统销售商提供的支持不够，或其它的决定改变了硬件和软件平台，拥有CASE系统的组织机构被迫作出替换的决定。另外，其它CASE系统为该特定组织提供了一个更合适的框架时，也可能作出废弃已有CASE系统的决定，用新系统取代正使用的系统。

不是简单地将CASE系统收藏起来，由一些其它系统取代，而是有一转移阶段，也许还要花几年时间，其间新旧系统要同时使用。在这个转移时期，用旧系统开发和继续维护的软件必须移到新的CASE系统。

这个问题的规模取决于要移植的软件总数，以及新老系统的兼容程度。废弃一个CASE系统的决定，也可能与废弃它所支持的软件的决定有关。当该CASE支持已成多余，中止该CASE系统又没有太多的花销时，就会废弃该CASE系统。

11.2 CASE 工作台

本节首先简要介绍了几种较为成熟的CASE 工作台，然后说明开放式CASE 工作台和封闭式CASE 工作台的特点，还描述设计、编程和测试工作台的结构和组成部件，最后介绍元-CASE，元-CASE 可以用来创建CASE 平台。

11.2.1 CASE 工作台概述

1. CASE 工作台的分类

如前所述，一个CASE 工作台是一组工具集，支持像设计、实现或测试等特定的软件开发阶段。将CASE 工具组装成一个工作台后工具能协同工作，可提供比单一工具更好的支持。可以实现通用服务程序，这些程序能被其它工具调用。工作台工具能通过共享文件，共享仓库或共享数据结构来集成。

工作台能支持大多数的软件过程活动。其中，像支持分析、设计、编程等软件过程活动的工作台比支持另一些活动的工作台更为成熟。工业界广泛使用这些较为成熟的工作台。

针对所开发软件的类别和应用领域的情况，可使用各种各样的工作台。工作台有：

①程序设计工作台，由支持程序设计的一组工具组成，如将编辑器、编译器和调试器等集成在一个宿主机上构成程序设计工作台供开发人员使用。

②分析和设计工作台，支持软件过程的分析和设计阶段。较为成熟的是支持结构化方法的工作台，现也有支持面向对象的方法进行分析和设计的工作台。

③测试工作台，趋于支持特定的应用和组织机构。常具有较好的开放性。

④交叉开发工作台，这些工作台支持在一种机器上开发软件，而在其它别的系统上运行所开发的软件。一个交叉开发工作台中，可能包括的工具有交叉编译器、目标机模拟器，从宿主机到目标机上下载软件的通信软件包，以及远程运行的监控程序等。

⑤配置管理(CM)工作台，这些工作台支持配置管理。如有版本管理工具，改变跟踪工具，系统建造(装配)工具等。

⑥文档工作台，这些工具支持高质量文档的制作。如有字处理器、单面印刷系统，图表图像编辑器，文档浏览器等。

⑦项目管理工作台，这些支持项目管理活动，有项目规划和质量、开支评估和预算追踪工具。

2. 开放式工作台和封闭式工作台

CASE 工作台可以支持一组相关的软件过程活动。这些活动从一个应用领域到另一个应用领域，从一个组织机构到另一个组织机构变化很大。因而，要求CASE 工作台应为开放系统。一个开放的工作台是这样的一个系统，或者提供控制集成机制，或者可裁剪(编程)，其数据集成或协议是公有的，而不是独立的。由于还没有被广泛接受的数据集成的标准，因而，大多数开放系统都采用基于文件集成的策略。

开放式工作台有如下优点：

① 如果有一个新的工具符合组织机构的特别需要，那么就能将这一工具加入到开放式工作台中。还可以用新的工具取代已有的工具。

② 工具输出的文件可以由一个配置管理系统来管理。

③ 工作台的功能能不断增强，工作台就可以不断发展。这样，一个组织机构可以最初只使用带几个简单工具的工作台。随着该组织内的工程师经验的增加，就可以将新的工具加入系统中。随着需要的增加，原先的工具能被功能更强大的工具取代。

④ 组织机构无需依赖于单个工作台供应商，而能从不同销售商处购买工具。能拥有不同支持商是很重要的。如果一个工具开发商不提供支持了，那么这最多只会影响该工作台的一部分，其余的工具还可以继续使用。

尽管开放式工作台优点很多。但许多CASE 工作台开发商还是决定提供封闭式系统。在封闭式系统中，系统的集成约定是该工作台开发商独有的。许多工作台都是封闭式系统，因为这允许更紧密地数据集成、表示集成和控制集成。出现在用户面前的工作台是一个一致的整体，而不是不同工具组成的工具箱。然而，第三方一般都不大可能为工作台增加工具，或增加功能。

自从 20 世纪 80 年代中期以来，花了很大力气来定义、实现和介绍开放仓库的标准。这些标准允许建立开放式工作台，工作台构件间能较紧密的集成。这些标准有针对 Ada 环境的 CAIS、采用面向对象方法的 ATIS，以及 PCTE。目前，PCTE 已被广为认可。然而，没有一个标准被广泛使用，大多数 CASE 产品还使用自己独有的方法来集成系统。

11.2.2 程序设计工作台

程序设计工作台由支持程序开发过程的一组工具组成。Ivic 在 1977 年介绍的程序设计工作台是第一个 CASE 系统。那时，将编译器、编辑器和调试器这样的软件工具一起放在一个宿主机上，该机器是专门为程序开发而设计制作的。像 Interlisp 这样的面向语言的编程系统也在 80 年代初被开发出来。

汇编器和编译器将高级语言程序转换成机器代码，它们是程序设计工作台的核心构件。在编译阶段生成的语法和语义信息也能被其它工具使用。这包括程序分析器、程序浏览器和动态分析器。程序分析器指明在哪儿定义和使用变量，程序浏览器显示程序结构，动态分析器创建一个动态程序执行轮廓。帮助程序员查找错误的调试系统也要用到语法树和符号表中的信息。

图 11.9 即为一个程序设计工作台。CASE 工具以图形框表示，工具的输入输出以矩形显示。在这个工作台中，工具通过抽象语法树和符号表集成，抽象语法树和符号表代表源语言程序的语法语义信息。

组成程序设计工作台的工具可能为：

①语言编译器：将源代码程序转换成目标码。其间，创建一个抽象语法树(AST)和一个符号表。

②结构化编辑器：结合嵌入的程序设计语言知识，对 AST 中程序的语法表示进行编辑，而不是程序的源代码文本。

③连接器：将已编译的程序的目标代码模块连接起来。

④加载器：在可执行程序执行之前将之加载进计算机内存。

⑤交叉引用：产生一个交叉引用列表，显示所有的程序名是在哪里声明和使用的。

⑥按格式打印：扫描 AST，根据嵌入的格式规则打印源文件程序。

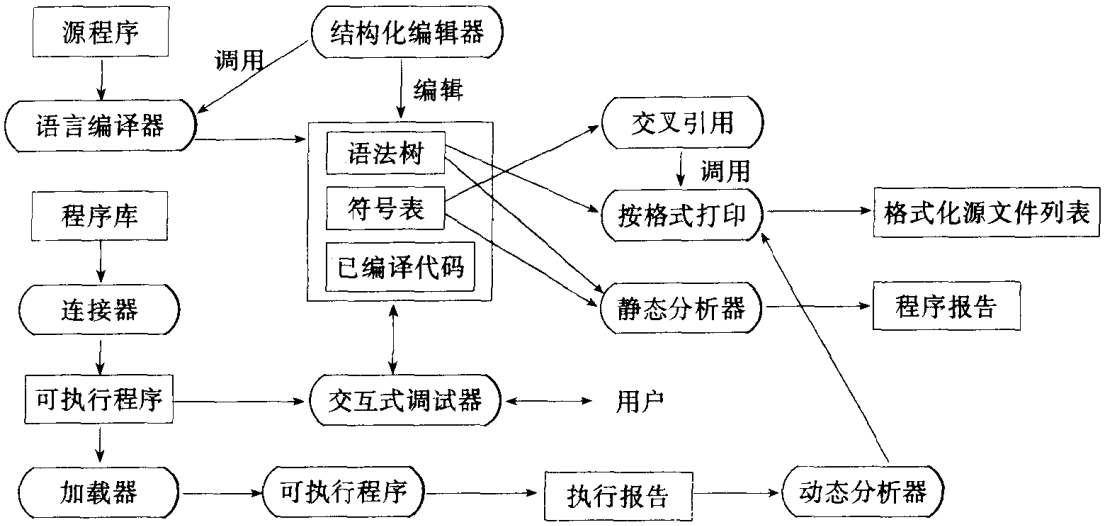


图11.9　程序设计工作台

⑦静态分析器：分析源文件代码，找到诸如未初始化的变量，不能执行到的代码，未调用的函数和过程等异常。

⑧动态分析器：产生带附注的一个源文件代码列表，附注上标有程序运行时每个语句执行的次数。也许还生成有关程序分支和循环的信息，还统计处理器的使用情况。

⑨交互式调试器：允许用户来控制程序的执行次序，显示执行期间的程序状态。

图 11.9 表明该程序设计工作台是利用语法树和符号表作为共享数据来进行工具集成的。逻辑上讲，提供各种各样工具的所有程序设计工作台中都采用该方法。然而，在开放式系统中，很难以这种方式来共享数据结构。开放式系统中的共享需要通过文件进行数据共享。很难创建复杂的互连结构的表示，而重新创建所要求的语法和语义信息会更容易更有效些。通常工具共享的数据是以程序源代码来相互交换的，那些程序源代码是对程序的语法树进行逆语法分析而生成的。

程序设计工作台也许能实现成像 UNIX 这样通用操作系统下运行的工具集合，也许和该程序设计语言编译器装配在一起。基于 UNIX 的系统是永远的开放式系统，这些系统由早期的 UNIX 程序员工作台演化而来。这些早期实践活动中产生的一些工具如(Make)已随 UNIX 系统一起出售了。

个人计算机上已有许多程序设计工作台。由于市场原因，通常这些工作台不作为工作台而作为包括有附加工具的语言编译器来出售。这些工作台通常是封闭式系统。在编译器和其它工具间，通过共享数据结构极紧密地集成。以这种方式出售的语言有 Basic，C，C＋＋，Pascal，Lisp 和 Smalltalk。

这些语言工作台通常包括一个面向语言的编辑器、编译器和调试系统。在执行过程中程序失败时，就初始化编辑器，并将编辑光标定位到导致失败的源程序语句处。而且，打开调试窗口显示失败时的程序状态。

这些工作台还可能包括程序查看系统，允许用户决定显示程序的方式和表示出的详细程度(图 11.10)。多程序视口有利于程序理解，它们给出了一个程序的全局结构画面。程序查看系统也许采用叠折结构，这样，比如说过程由过程头来表示，系统的使用者可以要求信息(如过

程体)显示或隐藏。

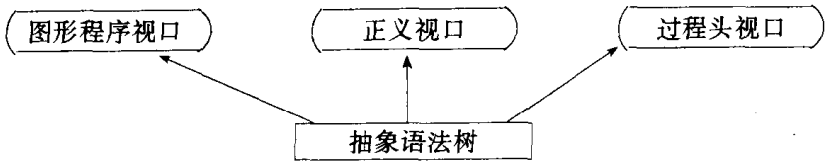


图11.10　多程序视口

强制式语言,诸如C,Ada或C++通常如图11.9所示,通过AST和符号表集成。第四代语言(4GL)使用另一种方法集成,现在4GL已广泛用于商业系统的实现。尽管命名为"语言",这些4GL实际上是程序设计工作台,因为它们常包括与程序设计语言无关的设施。4GL工作台通过数据库集成。图11.11给出了4GL工作台的一般组成。

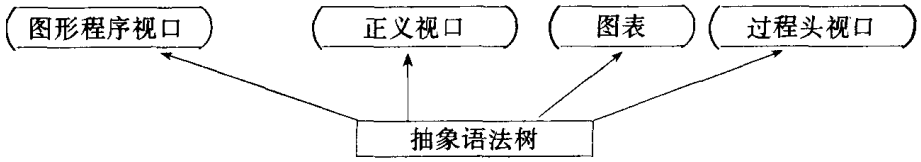


图11.11　一个4GL程序设计工作台

4GL工作台趋于产生交互式应用程序,该程序从数据库中抽取信息,将之提交给终端机或工作站用户,再随用户所做的改变来更新数据库。用户界面通常由一组标准表格或一个报表组成。

一个4GL工作台中可能包括如下工具:

① 诸如SQL的数据库查询语言,或是直接输入的,或是从由终端用户填写的表格中自动生成的。

② 一个表格设计工具,用于创建表格,数据通过表格输入和显示。

③ 一个电子报表,用于分析和操纵数字信息。

④ 一个报告生成器,用于定义和创建电子数据库信息的报告。

与强制式编程语言以程序为中心不同,一个4GL工作台是以数据库为中心的。数据库(而不是抽象语法树和符号表)是绑定工作台构件的集成"胶水"。4GL戏剧性地改变了商业系统的开发,大大降低了创造一个能运行的系统所需的时间。还可以进一步扩充4GL所产生的系统的类型,进一步提高生成可运行程序的自动化程度。一般来讲,用一个4GL工作台产生一个系统,大约只花费传统程序设计语言开发系统的10%—25%的时间。然而,这样的系统通常比用强制式语言产生的系统低效。因而对大型系统开发来讲,使用4GL并不很现实。

11.2.3　分析和设计工作台

分析和设计工作台支持软件过程的分析和设计阶段,在这一阶段,系统模型业已建立(例如,一个数据库模型,一个实体关系模型等)。这些工作台通常支持结构化方法中所用的图形符号。支持分析和设计的工作台有时称为上游CASE工具。它们支持软件开发的早期过程。程序设计工作台则称为下游CASE工具。

这些工作台也许支持特定的设计或分析方法,诸如JSD或Booch的面向对象分析。另外它们也可能是更为通用的图表编辑系统,能处理大多数通用方法的图表类型。面向方法的工作

台提供方法规则和指南。也可能进行一些自动图表检查工作。

一个分析和设计工作台可能包括的工具如图 11.12 所示。这些工具一般通过一个共享仓库集成，该仓库的结构是工作台开发商专有的，因而分析和设计工作台通常是封闭式环境。用户很难将他们自己的工具加入其中，也很难修改提供给他们的工具。

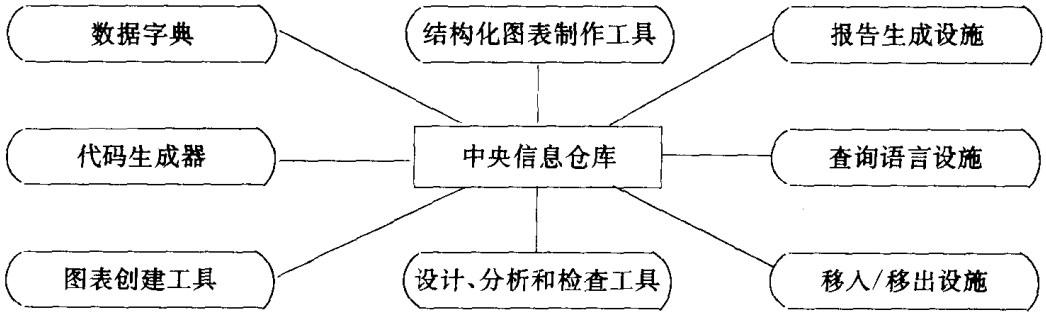


图11.12　一个分析和设计工作台

图 11.12 中显示的分析和设计工作台的构成为：

①图表编辑器，用来创建数据流图、结构图，实体关系图等。这些编辑器不仅是绘图工具，也能确认图表中出现的实体的类型。它们获取有关这些实体的信息，将这一信息存在中央仓库中(在有些工作台中称为百科字典)。

②设计分析和核实工具，进行分析，并报告错误和异常情况，这些也许和编辑系统集成，以便在早期开发阶段用户能追踪错误。

③仓库查询语言，允许设计者查询仓库，找到与设计相关的信息。

④数据字典，维护系统设计中所用的实体信息。

⑤报告定义和生成工具，从中央存储器中取得信息并自动生成系统文档。

⑥移入/移出设施，允许中央仓库和其它软件开发工具互换信息。

⑦代码生成器，从中央存储器获取设计信息，自动生成代码或代码框架。

现在，针对大多数结构化方法，都提供有分析和设计工作台。Chikofshy 和 Rubenstein 在 1988 年指出，利用这些 CASE 系统可将生产率提高 40%。他们还发现所开发系统的质量得以改善，所开发的系统的错误和不一致性减少了，所开发的系统更适合于用户的真正需要。

分析和设计工作台还有许多不足之处，这大多是由于这些工作台是封闭系统而导致的。它们有自己的存储管理系统。此类缺陷为：

① 移入/移出设施受限。尽管所有的工作台能移入和移出 ASCII 码文本形式的设计结果，大多数也愿提供图表的 Postscript 输出，但不支持其它的移入/移出格式。这在与其它工作台互换数据时会发生问题。

② 不能裁剪和修改一个设计方法。这样就不能用于特定应用或某类应用。用户通常不可能用自己的规则取代一个原有规则。

③ 工作台自己提供的配置管理系统可能与一个组织机构中使用的系统不兼容。这样，用设计工作台产生系统设计后，无法将设计结果交给组织机构中使用的配置管理和系统管理。

分析和设计工作台可能包含有代码生成器，它能生成用 Ada，C 或 C++编制的代码。由于设计不包括复杂的低级细节，因而设计工作台的代码生成器不能生成完整的系统。当然是尽

可能多地自动生成代码，但仍然需要手工编码。许多分析和设计工作台，如支持 HOOD 方法的 HoodNice 就采用的这种自动代码生成与手工编码相结合的方法。

使用代码生成方法后，维护阶段要求系统改变时，可以先改变设计，再重新生成代码。如果直接进行代码维护，就不可能避免设计和编码的不一致性。

有些分析和设计工作台支持开发商使用程序。开发平台和应用平台相同。因而用像 Ingres 或 Oracle 这样的数据库系统来实现共享仓库。这些工作台包含有大多数 4GL 程序设计环境的设施，因而，一个设计不是生成传统程序设计语言程序，而是生成一个 4GL 数据库语言程序。

这种程序设计语言隐藏在标准图形格式和脚本之后，因而无需显式地编程。在分析、设计和实现之间有无缝连接，在软件开发过程中没有单独的实现阶段。这导致了应用程序快速原型生成，避免了 4GL 的一些维护问题。然而，用这种方法所创建的应用程序时常包括冗余信息。因而代码量很大，执行速度还可能很低。主要由于其效率低，因此这种方法不适于开发大型商用系统或事务处理系统。

11.2.4 测试工作台

测试是软件开发过程较为昂贵和费力的阶段。其结果，在最早先的第一批软件工具间就开发了测试和调试工具。这些工具现在提供了各种设施，使用它们极大地降低了测试和调试阶段的费用。

测试一般支持特定的应用和组织机构。因而，封闭式测试工作台的市场不大。公司更趋于组装购进的和本地实现的测试工具来创建自己的测试工作台。因而，测试工作台永远应为开发系统，可以不断演化以适应被测试系统的需要。

图 11.13 显示一个测试工作台包含的一些工具，以及这些工具之间的互操作情况。

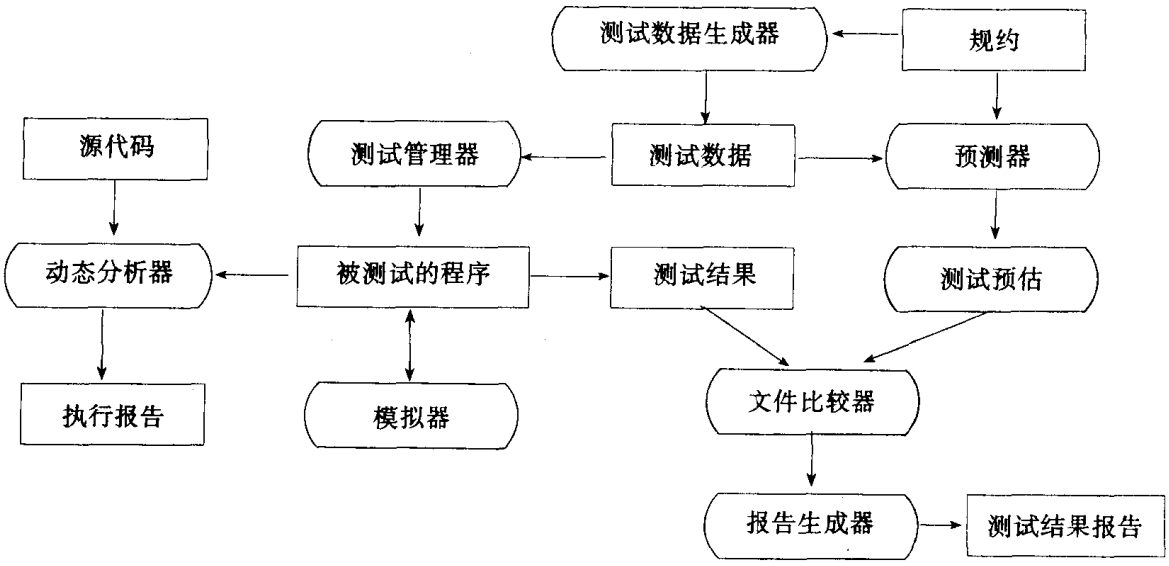


图11.13 一个测试工作台

测试工作台包括的工具有：

① 测试管理器 管理程序测试的运行和测试结果报告。这涉及对测试数据的跟踪，对所

期待结果的跟踪，对被测试的程序的跟踪等等。

② 测试数据生成器　生成被测程序的测试数据。这可能是从一个数据库中选择数据，也可能是使用模式来生成正确格式的随机数据。

③ 预测器　产生对所期待测试结果的预测。预测器所采用的或是以前的程序版本，或是原型系统。背靠背的测试涉及并行运行预测器和要测的程序，将强调显示它们输出的不同之处。

④ 报告生成器　提供报告定义，提供测试结果的生成设施。

⑤ 文件比较器　比较程序测试的结果和以前测试的结果，报告它们之间的差别。在回归测试中，比较特别有用；回归测试，比较的是新版本和旧版本的执行结果。这些结果中的差异指示了系统的新版本存在的潜在问题。

⑥ 动态分析器　将代码加到一个程序中以计算每条语句被执行的次数。运行完测试之后，将生成一条执行轮廓线，显示每条语句被执行的频率。要设计测试用例，使得程序中所有语句都将至少被执行一次。

⑦ 模拟器　可能提供各种不同的模拟器。目标模拟器是脚本驱动的程序，模拟多个同时进行的用户交互。I/O 模拟器的使用意味着事务次序时标是可重复再现的。这在测试实用系统时，被测系统如果带有微小的定时错误，这一功能就特别有用。

大型系统的测试需求依赖于要被开发的应用程序。其结果，总要更改测试工作台以适应每个系统的测试计划。要求修改的例子如下：

① 要为用户界面模拟器编写脚本，要为测试数据生成器定义模式，也可能要定义报告格式。

② 如果不能将以前的程序版本作为预测器输入用例，就有可能不得不手工准备所期待的测试结果集。

③ 要编写特定目的的文件比较器，这就要有文件测试结果的结构知识。

要花费大量的时间精力来创建一个可比较的测试工作台。图 11.13 所示的完整测试工作台仅用于要开发相当昂贵的系统之时。在这种情况下，整个测试的费用会上升至整个开发费用的 50%，因而通常要论证测试工作台修改费用的合理性。

11.2.5　元-CASE 工作台

不同语言的程序设计工作台有许多共性。尽管细节不同，但它们都会用到抽象语法树和符号表。所要求的操作很相似，例如，从语法树上增加或移开一个结点，以特定方式显示一个节点。

由于这种相似性，就可能编写一个程序来生成针对特定语言的程序设计工作台。这和“编译器的编译器”概念类似，后者给定一个语言的语法和语义信息，来生成该程序设计语言的编译器。工作台生成器获取有关语言的语法和语义信息，创建一个针对该语言的工作台。

早在 20 世纪 80 年代初就认识到这种可能性，第一代工作台生成器的典型代表是 Mentor 系统、分析程序生成器和 Gandalf 系统。这些系统包括一个类属环境，此环境是由表示特定程序设计语言信息的一组表格驱动的。这些表格产生于带属性的语言语法定义，而语言语法定义所带的属性是语义信息（图 11.14）。

这些环境生成器是现在所谓元-CASE 工具的首批实例。元-CASE 工具被结合到工作台

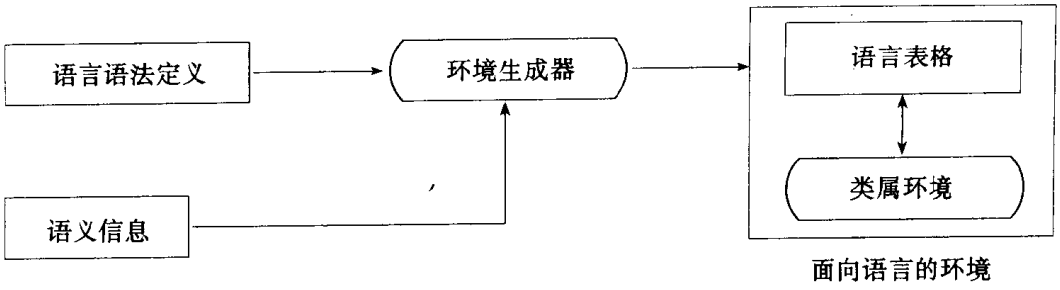


图11.14　环境生成器

中，以用于创建其它的CASE工具和工作台。

面向语言（或更精确地说，面向语法）的程序设计工作台的开发是相当直接了当的，因为程序设计语言有定义良好的语法和“可接受”的语义。而分析和设计工作台常用到的图形概念就不是这样的。其语法常是非形式化定义的，语义嵌在与设计和分析方法相关的规则和指南中。

20世纪80年代中期的研究解决了这一问题。如Beer在1987年所述，他是通过产生一种能用于描述设计方法的语言来解决的。这一语言定义了方法类别、与方法相关的规则等。与利用程序设计语言生成程序设计工作台类似，使用这种方法描述语言可以生成设计编辑器的CASE工具，所用的一般规则和图11.14所演示的模型相似。对方法的描述进行编译，创建一组表格供类属编辑系统使用。由于方法一般用到许多不同的图形概念，如数据流图，结构图等，因而为每种概念产生单独的表格。所有的设计都使用一个编辑器，呈现给用户的是一个一致的用户界面。当对一个特定类别的图表进行编辑操作时，就将对应的方法描述表格动态地和该编辑器相连（图11.15）。

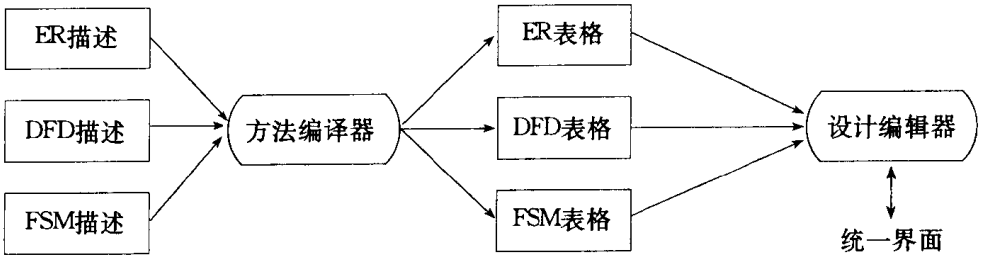


图11.15　一个多概念的设计编辑器

图11.15说明了如何根据实体关系图、数据流图和有限状态机的描述，来创建一个能处理所有这些图表种类的一个编辑系统。由于对所有图形编辑操作提供一个共同的界面，因而这一方法具有极好的表示集成性。

定义特定工具的特性，根据这些特性剪裁类属系统。元-CASE就是基于这样的类属系统模型的。除图表编辑系统外，元-CASE工作台的其它工具还有：

① 通用仓库（如PCTE所采用的），模式描述和编辑工具；

② 创建结构编辑器的工具；

③ 代码生成器和模式定义工具，后者定义特定的设计结构，前者按所定义的结构生成代码；

④ 一个表格编辑器和与仓库相连的报告生成器。

Alderson 1991年描述了一个称为Toolbuilder的元-CASE工作台，它可产生支持方法的

CASE 工具。这一系统基于这样一个事实:所有的设计方法都把设计表示成带属性的有向图,因而可以开发操纵这些图形的类属系统。该CASE 工作台所支持的方法有五个方面:

① 供数据获取和输出生成用的数据模型;

② 定义框架模型,该框架模型生成数据模型视图;

③ 为每个图形框架用的图示符号;

④ 供每个正文框架用的文本表示;

⑤ 报告的结构。

ToolBuilder 系统实现了许多不同的工作台,它也可生成自己。Alderson 报告说在几个星期内可产生原型工作台。这样,可以快速开发适应特定应用需要的新的方法支持和相关CASE 支持系统。

元-CASE 工具降低了生成CASE 工作台的费用,但是,可能更为重要的是它们解决了方法不易改变的问题。组织机构也许希望裁剪设计方法,以适合他们自己的需求,但修改支持工具通常是不可能的。然而,如果用一个元-CASE 系统来制造这些工具,方法改变的费用就相当低了,只需简单地修改该方法定义,就能生成一个新的工作台。

11.3 软件工程环境

本节首先介绍软件工程环境的概念及其基本支持机构;然后讨论软件工程环境的优点和缺点;还描述了一个软件工程环境的框架参考模型,介绍与参考模型相关的服务;最后简要介绍了 PCTE 这一广泛推荐的关于软件工程环境的框架标准。

11.3.1 软件工程环境概述

1. 发展简介

随着软件工程的兴起,20 世纪 70 年代中期开始出现了支持程序开发、维护的工具。这些工具大都将重点放在建立程序的相关文档上,其基本出发点是将文档建立过程也作为系统开发过程,而不是在系统开发结束后再补写文档,从而保证了文档的可信度。其中典型的工具是ISDOS(Information Sytstem Design and Optimization System)系统中的 PSL/PSA,它主要用于系统报表及文档的生成。随后,工具箱(Toolkits)的思想开始出现。工具箱中有很多软件工具,这些工具间相互独立,不依赖某一方法论,可以为用户单独使用。工具箱中的工具往往具有统一的用户命令界面,工具采用统一的数据交换方式。波音公司的 AGRUS 软件开发系统就属于工具箱。工作台的概念也开始使用。

在 70 年代末期开始出现交互式软件开发系统,它提出了用户友好、方便的图形用户界面的思想。其中最典型的是 Xerox 公司的面向对象语言 Smalltalk 程序设计系统、APPLE 公司的 Macintosh 等。

80 年代,支持 Ada 语言的 APSE 环境模型提出后,"程序设计环境"、"软件开发环境"等有关环境的术语广泛使用,使得进入 80 年代后,环境的研究成了热点。同时,支持图形设计方式的软件工具开始大量涌现,包括支持结构化分析设计方法的工作台,如支持数据流图(Dataflow Diagrams),模块结构图(Modular Structure Chart)、状态变迁图(State-transition Diagrams)等的编辑及分析的工作台。这一时期的软件工具的特性是:①对结构化方法的自动

化支持;②对单个系统分析员的支持;③对软件开发过程的部分覆盖;④对分析效率及确认能力的改善。另外,集成这些工具为一体的软件工程环境也得到发展,出现了以环境信息库为核心的软件工程环境。如 McDonnell Douglas1989 年开发的 ProKit * WORKBENCH(PKWB),Interactive Development Environments(IDE)开发的 Software through Pictures(StP)。

总之,可以认为软件工程环境包含有一组软件工具,这些工具是按照一定的方法或模型组织起来的,这些工具支撑整个软件生存周期的各个阶段或部分阶段。其宗旨是为计算机软件的生产提供计算机辅助手段,改变长期以来软件产品的手工式生产方式,以提高软件产品的生产率,降低软件的成本和改善软件的质量。有时我们也称软件工程环境为 CASE 环境。软件工程环境一词近义词很多,如软件开发环境(SDK)、软件支撑环境(SSE)、程序设计支撑环境(PSE)等等。

80 年代后期,典型软件工程环境的显著特点有:①采用环境信息库,工具围绕信息库集成;②支持软件开发模型及软件开发方法,例如瀑布模型和结构化方法;③集成机制的研究有了较大的发展,出现了集成型软件工程环境。集成型软件工程环境是把支持多种软件开发方法和过程模型的软件工具集成到一起的软件工程环境。它由环境集成机制和工具集两大部分组成。环境集成机制在统一的接口规范和工具结构模型基础上可形成开放的工具插槽,使工具可以像插件一样集成到环境之中。NIST/ECMA 提出了集成型软件工程环境参考模型。ESPRIT 的 PCTE(Portable Common Tools Enviroment)采用了这个参考模型,1990 年成为 ECMA 的标准。但软件工程环境的国际标准尚未形成。

企业 CASE 的概念出现于 80 年代末和 90 年代初。企业 CASE 一般涉及一个大规模的仓库,该仓库具有足够的健壮性和实力为整个大组织开发提供资源,并起到协调者的作用。

90 年代开始出现支持面向对象方法与技术的软件开发环境。受到学术界和产业界的重视。这样的环境有 EasySpec 的 Object Plus,它是基于 Windows 的集成型软件工程环境,支持面向对象方法。11.4 节介绍的青鸟系统也是支持面向对象的软件工程环境。

2. APSE 模型

CASE 工作台,如前所述,是以不同方式支持软件过程不同阶段的"自动化岛屿"。每个工作台管理它自己的数据,使得很难对所有软件过程输出提供一致的配置管理。在一个工作台生成供另一个不同工作台复用的信息是很难的,或是不可能的。

软件工程环境(SEE)就是要解决这种困难,支持所有或大多数软件过程活动。这样的一个环境概念首先是由 Buxton 于 1980 年介绍给公众的。Buxton 在美国国防部支持下,提交了一组支持 Ada 程序设计环境(APSE)的需求。Buxton 提出了如图 11.16 所示的 APSE 模型。

Buxton 认识到 1980 年不可能建立一个完全的 APSE。他提议采用增量式开发方法来开发基于三个功能级别的环境:

① 一个核心 APSE,它提供环境的基础机构,对操作系统进行扩充。Buxton 建议这种核心 APSE 采用标准化的核心,并且有一个公共工具接口。从而在这一相同的核心之上建造不同的环境产品。

② 一个最小的 APSE,它基本上是一个程序设计工作台。在 1980 年所能提供的硬件平台上(仅带文本终端的分时系统),这可能是 APSE 提案所能现实提供的全部了。

③ 以增量式开发的一个完整的软件工程环境,当支持其它过程活动的工具可提供时,可将这些工具加入最小的 APSE 以扩充其功能。

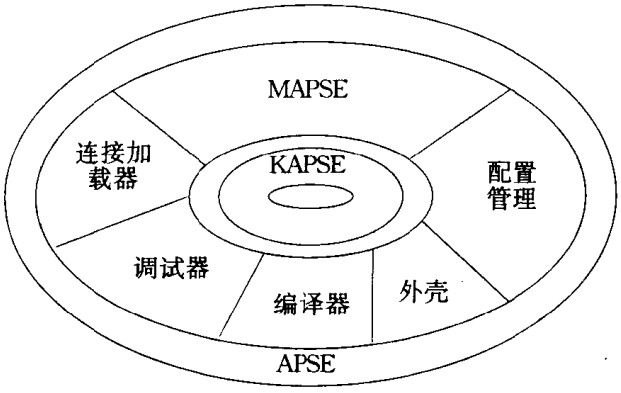


图11.16 APSE模型

这些提议极赋远见卓识，并且极大地影响了软件工程界。标准化内核的观点导致CAIS和PCTE等的研制。许多政府组织和大公司在20世纪80年代中期开始研究开发APSE或更通用的软件工程环境。

3. 软件工程环境分类

软件工程环境可从以下几种角度分类：

① 按软件开发模型及开发方法分类，有支持瀑布模型、演化模型、螺旋模型、喷泉模型以及结构化方法、信息模型方法、面向对象方法等不同模型及方法的软件工程环境。

② 按应用范围分类，有通用型和专用型软件工程环境。其中专用型软件工程环境与应用领域有关，故又可称为应用型软件工程环境。

③ 按开发阶段分类，有前端开发环境（支持系统规划、分析、设计等阶段的活动）、后端开发环境（支持编程、测试等阶段的活动）、软件维护环境和逆向工程环境等。此类环境往往可通过对功能较全的环境进行剪裁而获得。

4. 软件工程环境的特征

一个软件工程环境一般有如下特征：

① 仓库：仓库是软件工程环境最重要的特征，其同义词有字典、数据库、库等。仓库保 存着正被定义的目标系统的描述，这些描述是对软件工程环境所支持的软件开发各阶段的工作产品的描述。在实现上，库可以是集中式的，也可以是分布式的。

② 工具的集成：软件工程环境提供了一组集成化的软件工具，通过这些工具操作和分析系统描述的结果。

③ 用户友好的界面：软件工程环境可以运行在主机、服务器系统、工作站或PC机上，甚至把功能分布在不同的机器上。为方便用户，应提供一致的图形界面。

④ 提取信息的能力：软件工程环境的用户在整个开发过程中，需要以不同的格式提取信息。

⑤ 分析的能力：把用户的工作产品以集成的方式存储在软件工程环境中后，在分析当前收集或定义的数据的基础上，进行一致性检查和完整性分析。

⑥ 可裁剪性和可扩展性：为了满足用户和开发组织的不同需求，软件工程环境必须是可裁剪的和可扩充的。

⑦ 项目控制和管理：软件工程环境应有助于良好管理的实施。如帮助管理构造系统描述，收集信息并进行度量。

⑧ 方法学的支持：软件工程环境可以支持某一种或多种系统开发方法，可供项目和开发组织使用。

5. 软件工程环境的基本功能

较完善的软件工程环境通常具有如下功能：

① 软件开发的一致性及完整性维护；

② 配置管理及版本控制；

③ 数据的多种表示形式及其在不同形式之间自动转换；

④ 信息的自动检索及更新；

⑤ 项目控制和管理；

⑥ 对方法学的支持；

等等。

11.3.2 集成环境

术语“环境”可用于极大范围的支持系统，其范围从支持单个语言程序开发的简单系统到能对使用许多不同语言和设计方法的极大型系统。如前所述，本书使用这一术语来描述支持所有(或许多)软件过程活动的CASE系统。

更为特别的，一个软件工程环境可定义如下：

一个软件工程环境(SEE)是软件和硬件的集合，软硬件集成在一起支持整个或绝大多数软件过程活动，包括规约到测试和系统发布等等。

一个SEE不同于CASE工作台的关键特性是：

① 环境设施是集成的。理想状态下，环境应支持5个方面的集成，即平台集成、数据集成、表示集成、控制集成和过程集成。

② 环境是小组使用，而不是个体开发，提供支持所有活动的配置管理。对系统文档和代码的众多版本以及其依赖关系进行管理，这是大型长期系统中特别费劲的活动。

③ 支持广泛的活动。结果，SEE可能包括支持规约、设计、文档、制作、编程、测试、调试等的工作台。

软件工程环境隐藏了巨大的复杂性。软件过程极为丰实，使得支持整个或大多数软件过程的环境是大型复杂的软件系统。

软件工程环境一般通过提供项目信息的共享仓库来支持集成。所有工具都与该仓库打交道，通过该仓库来交换信息。一个工具的输出成为另一个工具的输入。尽管大多数工具互操作是可预知的，但还是有许多种不可想象的工具组合形式。通过仓库交换信息解决了CASE工作台的最大问题，即工具和其它工具互换信息的问题。项目管理可访问项目信息，管理工具可使用项目过程中收集的产品数据。

配置管理的费用在软件工程环境中降低了，所有项目文档都存放在一个位置，可以自动管理这些文档间的连接。如果环境仓库支持细粒度对象，配置管理可以扩展到像单个变量声明这样的实体。这简化了可跟踪工具的开发，在环境不得不发生变化时，可使用这些工具确定相关的依赖情况。

为了能让软件工程环境可根据不同项目需要来提供不同的支持，软件工程环境必须能容纳广泛的CASE工具和工作台。也必须能按需增加新的设施。这意味着一个软件工程环境可

认为是一组服务的集合，那些服务能为支持终端用户的设施所用。提供这些服务的既可以是软件工程环境运行其上的平台，也可以是环境框架。图 11.17 显示了这三层模型。

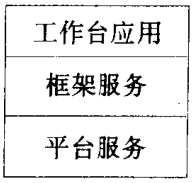


图 11.17　一个软件工程环境的层次模型

在这个模型中，一个软件工程环境是由使用环境服务的一组集成的 CASE 工作台组成的。这些服务既可由环境运行于其上的平台提供，也可以由环境框架提供。框架服务可类比于 APSE 的环境核心。

Brown 等在 1992 年指出，根据环境用户角色的不同，对这一软件工程环境模型有许多不同的看法。其中最重要的有：

①应用软件开发人员认为环境是支持软件过程的一组工作台集。他们主要关心工作台应用层。

②软件工程环境集成人员把环境看作一组通用服务和工具的集合，其中的工具是指那些集成在特定背景中的，用以创建高效支持环境的工具。他们主要关注图 11.17 的工作台和框架层之间的边缘部分。

③工具或工作台开发人员将环境看作一组通用服务的集合，那些服务用于将他们的工具与其它别的工具一起集成到环境中。他们主要关注图 11.17 的中间层。由于他们不知道组成环境的别的工具的情况，因而他们不关心图 11.17 中的工作台应用层。

④ 框架开发人员把环境看作一组服务的集合，必须在某些宿主机系统中实现那些服务。他们关心如何使用机器的设施来提供有效的服务实现。他们主要关注图 11.17 的中间和最底层之间的接口。

显然，一个环境提供的服务很重要，这些服务支持工具的实现和集成。下面介绍图 11.17 的服务层。

11.3.3　平台服务

一个软件工程环境运行其上的平台称为软件工程环境的宿主机系统。某些情况下，使用软件工程环境开发的软件运行在相同的平台上，但是，有许多情况，所开发的软件将分发到可能具有完全不同的构架和操作系统的一些目标机系统上。图 11.18 说明了这种宿主-目标开发方式，宿主机上所开发的软件分派到不同的目标机上。

采用宿主-目标开发方式的理由如下：

① 某些情况上，正开发的应用软件是为一个没有软件开发机制的机器所用的。这大多数是为专用计算机开发实时系统。这些计算机甚至还没有一个操作系统，只有一个简单的实时执行机制。

② 目标机也许是面向应用的(例如一个向量处理器)，不适于支持软件工程环境。

③ 目标机正在使用之中，专门运行一个特定的应用(如事务处理系统)，因此不能用该目

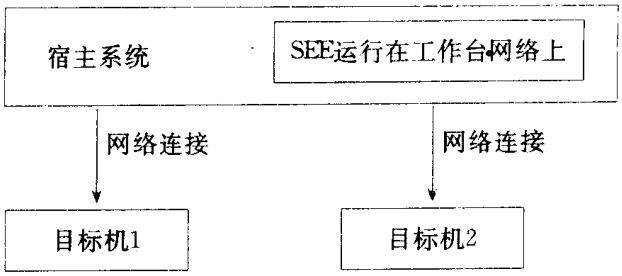


图11.18　宿主－目标开发方式

标机进行软件开发。

①文件服务：一般来讲，作为软件工程环境宿主机的平台提供如下服务：

支持文件命名、创建、存储、删除，并将文件按目录结构组织起来。正常情况下，文件存储在一个或多个文件服务器上，可为网络上所有机器访问。

②进程管理服务：用于创建、开启、停止和挂起在网络计算机上运行的进程。

③网络服务：用于把进程和其相关数据从一台计算机移至另一台计算机。

④通信服务：用于与本机构其它位置上的其它的计算机通信。如果目标机与网络相连，这一服务必须能支持将程序下载到这些机器上。

⑤窗口管理服务：允许在用户显示器上创建、移动、删除窗口，改变窗口大小等等。

⑥打印服务：允许将环境的信息打印到纸上，或打印到其它永久性媒体诸如 CD-ROM 或缩微卡片上。

一般来讲，环境平台是由异构分布式计算机组成的。这也许包括不同类型的计算机（例如，UNIX 工作站和 PC 机），来自相同制造商运行不同操作系统版本（如 Solaris 2.1 和 Solaris 2.4）的 UNIX 工作站，以及来自不同制造商的 UNIX 工作站。

在拥有许多不同计算机的大型组织中这种异构是不可避免的。新工作站通常只运行最新版本的操作系统。这还得让运行于老版本的操作系统之上的已有系统运行起来。因而，相同操作系统的不同版本可能在相同网络上并发运行。进一步来讲，同一组织机构的不同部分可能有不同的计算需求，因而购有不同类型的机器（如，图形工作台）。显然他们希望使用这些来运行软件工程环境，而不是专为软件开发购买别的计算机。

11.3.4　框架服务

软件工程环境中框架服务扩充了环境宿主机提供的服务集，通常框架服务是利用平台服务来实现的。这些服务专用于支持 CASE 工具或工作台的集成。

理解框架服务最好的途径是在一个软件工程构架的参考模型比较易于理解的背景下来理解它们。构架参考模型提供了一个比较不同构架的基础。软件工程环境的参考模型最早由欧洲计算机制造厂商协会（ECMA）提出的，现在已为美国国家标准和技术局（NIST）采用。“正式的”模型在 1997 年度 ECMA 的 NIST/ECMA 报告中给出。但对该模型的描述是由 Brown 等在 1992 年给出的。Brown 用这个模型比较了不同的环境框架提案。

图 11.19 显示了该模型的结构（已以“更粗”模型而闻名）。关键在于，该参考模型确定了一个环境应该提供的五类服务集。它也提供了使用这些服务的工具和工作台的“插入”设施。其结果是，根据所使用的软件过程和开发的软件类型，可以用不同的工具和工作台来配置软件。

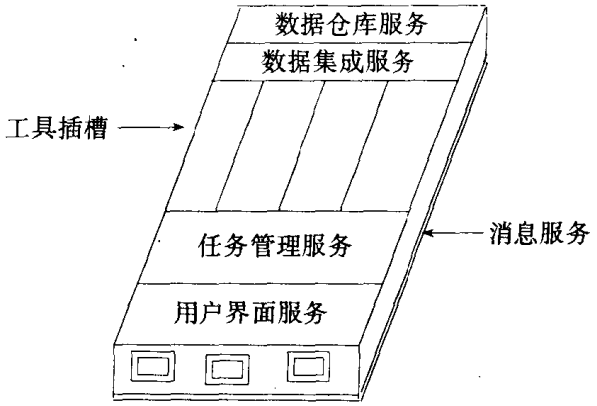


图11.19 SEE的参考模型

一个软件工程环境提供的服务可能为：

① 数据仓库服务　对数据项及其关系进行管理。

② 数据集成服务　提供了对数据项成组或配置进行管理的设施。它们支持配置命名，支持建立配置间的关系。这些服务和数据仓库服务是环境中数据集成的基础。

③ 任务管理服务　提供了软件过程模型定义和实例化的设施。它们支持过程集成。

④ 消息服务　提供了工具到工具，环境到工具，以及环境到环境通信的设施。它们支持控制集成。

⑤ 用户界面服务　提供用户界面开发设施。它们支持表示集成。

1. 数据仓库服务

数据仓库服务提供了命名实体、管理实体、建立实体间关系的基本设施。表11-2介绍了参考模型中所确定的数据仓库服务。

表 11-2　数据仓库服务

服务	描述
数据存储	支持实体的创建、读、更新和删除
关系	定义和管理环境实体间的关系
命名	支持实体命名。这是赋予实体的唯一的标识符
定位	在工作站网络上分派实体，因而有像移动、拷贝、重复等相关操作
数据事务	支持原子事务，允许发生失败事件时的数据库恢复
并发	支持多个事务处理同时进行
进程支持	提供诸如开启、停止、挂起等进程操作
文档	支持实体的脱机存储和恢复
备份	发生系统失败事件时，支持数据恢复

这些数据仓库服务通常是由对象管理系统(OMS)提供的。已实现的大多数对象管理系统是基于实体-关系模型的，并提供一些反映软件过程需要的内置实体和关系类型。

图11.20显示了在OMS中，代表过程或函数的实体是如何相连的。图中假定环境使用一个数据的实体关系模型。实体、属性和关系被分类，而这一类型信息为软件工程环境工具所用。

在该例中，说明了用不同程序设计语言编写的4个例程A，B，C，D的情况。每个都有不同的属性，如所使用的开发语言、创建者、其状态和它存储在哪个库中。这只是一个示意图，未显

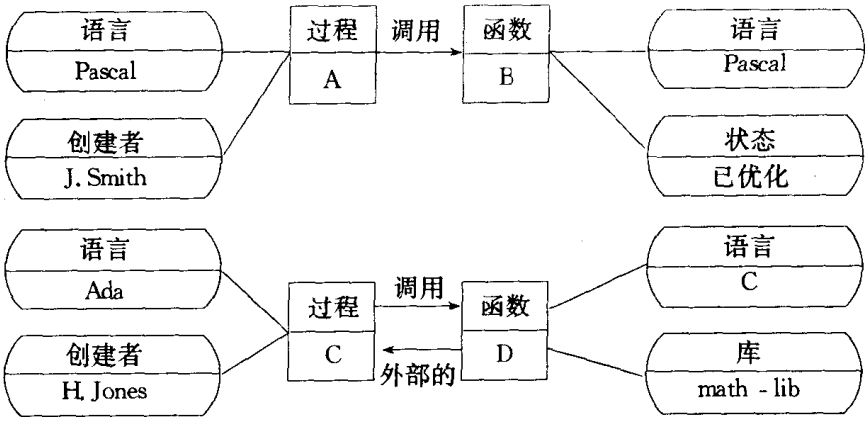


图11.20 OMS中分类的实体、属性和关系

示对象的所有属性。

该模型中显示过程A调用函数B，两者都是用Pascal写的，对B进行某种程度的优化。过程C是用Ada写的，调用函数D。这是一个外部函数，用C语言编写，存在称为math-lib的库中。可以提供浏览器，来浏览OMS中的对象及其属性，以及对象之间的关系。

一个对象管理系统或数据库的粒度取决于能有效存储和操纵的实体的最小尺寸。如果能被有效操纵的最小实体尺寸是文件，那么所管理的关系类型就允许文档被连接。然而，在这些文档中不能建立实体间的关系。相反，如果只有几字节长的细粒度实体能被管理，那就能使用更丰富的关系集。例如，OMS就记录程序中变量声明间的连接及其使用情况。

检索数据时，细粒度系统比粗粒度系统需要更多的数据库查询。这就意味着这些系统的操作性能要比粗粒度系统差。这样一来，大多数数据仓库提案都推荐使用粗粒度而不是细粒度数据管理。

2. 数据集成服务

框架中提供的数据集成服务集扩展了基本数据仓库服务。数据仓库服务是通用的，而与数据集成服务相关的操作显然是支持软件开发的。表11-3描述了软件工程环境参考模型中提议的数据集成服务。

表11-3 数据集成服务

服务	描述
版本	支持实体多版本管理
配置	将实体分组，按组进行配置命名；并作为一个完整的实体来管理
查询	提供访问和更新版本的服务
元-数据	提供模式定义和管理设施
状态监控	提供触发设施，允许当达到特定数据库状态时，初始化特定操作
子环境	能定义和管理环境的数据和操作的一个子集，并将之当作一个单独的命名的环境
数据互换	支持从环境中移入和移出数据

在一个软件工程环境中，配置管理是非常关键的。广泛使用的是基于文件的配置管理工作台，其缺点是所管的实体只能通过结合在工作台中的数据松散连接。当配置支持作为一个服务，就连接仓库中所有的项。可以创建和管理版本和版本集，跟踪变化，在环境实体间建立可追

踪的关系。

元-数据服务和子环境服务允许创建带自己的数据和关系类型的本地环境。元-数据是关于数据的数据，而元-数据服务能定义新的实体和关系类型，能查询所定义的类型。子环境服务允许划分环境数据。子项目可有自己的环境。因而它们同时可以与其它正进行中的项目相孤立开来。

状态监控服务允许通过数据状态的改变来启动特定的动作。例如，当创建一个实体的新版本时，就可以触发操作，自动通知该实体的所有用户已可用该实体的新的版本了。通过数据互换服务，一些工具能维护自己的仓库，还能将信息从它们局部存储区中移进和移出。

3. 任务管理服务

任务管理服务支持环境中的过程集成。表 11-4 介绍了参考模型已定义的服务。在这样的上下文中，术语"任务"意指某些过程中原子活动单位。

本质上讲，任务管理服务提供定义和实施(执行)过程模型的操作。这些模型包括诸如活动、角色分派的实体。必须实施这些模型提供支持软件开发过程的活动。可能将活动赋予角色，当某事件(例如一个任务开始)发生时激发活动。允许记录活动历史，记录资源使用情况以及维护资源等等。

任务管理服务可能是参考模型中定义最不完善的服务。1992 年 Brown 将几个框架提案与参考模型比较时，发现这些系统都不能很好地支持过程管理服务。

表 11-4　任务管理服务

服务	描述
任务定义	提供定义任务的机制，该机制包括前置条件和后置条件，输入和输出，需要的资源和任务中涉及的角色
任务执行	提供支持任务执行的设施。这也许包含用一个过程编程语言来描述任务的交互操作
任务事务	提供对事务的支持，这些事务在相当一段时间内与一个或多个任务执行有关。应能做到无需将系统退回任务开始执行的状态就能从失败中恢复
任务历史	提供设施来记录任务的执行，查询以前的执行
事件监控	支持事件或引起某任务执行的触发的定义
查账与记账	记录做了什么，以及环境中哪些资源被使用
角色管理	提供定义和管理环境中角色的设施

4. 消息服务

消息服务允许工具和框架服务通信。在软件工程环境参考模型中只定义了两种消息服务。

① 消息发派：这一服务支持工具到工具、服务到服务、框架到框架之间的消息传递。可能允许点到点(即，直接)消息互换，允许发送到所有工具和服务的广告消息，允许由一些代理来确认哪些工具和服务应接收该消息，这样可进行消息的多重传递。相关的操作有发送、接收、应答等。

② 工具注册：允许一个工具或服务作为某种类型的消息的接收者登记到消息服务器上。

在许多商用产品上，如 HP 的 SoftBench 已实现了这个简单的模型。它提供了一个分布式网络上的控制集成。一个工具或服务能在某个中央代理处登记，说明所感兴趣的消息类型。当另一个工具或服务希望通信时，它向消息发派服务发送消息。这确定了网络上该消息潜在的接

收者，并将消息发送给它们。

5. 用户界面服务

用户界面服务支持表示集成。软件工程环境参考模型的开发者没有定义一组新的用户界面服务集，而是提议用户界面服务应基于OSF的分层用户界面模型(图11.21)，该模型是在X-Window的早期工作基础上形成的。

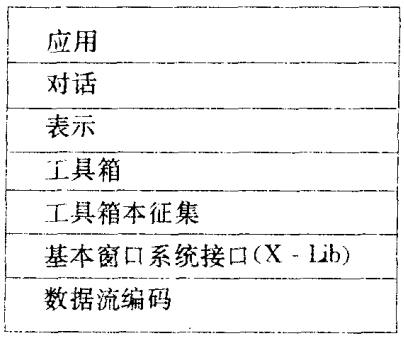


图11.21 用户界面参考模型

从下往上看，这个用户界面参考模型的头两层提供了基本的物理支持，包括屏幕和输入设备处理，一组基本的图形原语(X-Lib)。接下来的两层建立在这些原语之上，提供了一组能用于表示层创建用户界面的界面对象。在这一级别上，Motif 工具箱已作为事实上的标准而出现。这些较底层的服务，可能属平台服务而非框架服务。

这级别之上提供的支持较为模糊。表示层支持在设计应用程序界面时直接使用工具箱中提供的图形对象。表示层可能有格式设计工具、用户界面布局工具等。对话层支持界面操作的同步，因而可能包括用户界面管理系统(UIMS)，还包括对话规范说明语言。最后，应用层与应用程序和用户界面间的通信有关。

与软件工程环境参考模型的其它服务不同，用户界面服务并未真正提供一个基点来比较和评估框架提议。软件工程环境参考模型假定所有的框架都支持X/Motif 和一些相关的支撑工具。然而，这个假定并非事实。

软件工程环境参考模型的设计者冒然假定大多数软件工程环境都将运行在UNIX 或一些相似的操作系统上。这是不现实的。如个人计算机上运行的是不同的操作系统，现在它们的功能已强到可以在网络环境中作为一个节点来使用了。个人计算机上运行一些别的窗口管理系统，如MS Windows，这些就与软件工程环境参考模型的假定不相符。

6. 工具

环境框架已为工具的实现提供了一组通用服务。然而，这些服务并非为所有工具使用。一些工具提供了自己的相应的服务，也应能将之与环境集成。因此，环境在平台和框架级都应提供集成工具的设施。

软件工程环境中，有三个工具集成的级别，如图11.22 所示。

① 集成工具：这些工具用框架服务来管理它们所有的数据，其数据结构存储在对象管理系统中。

② 半分离工具：这些工具与框架服务的集成不怎么紧密。它们管理自己的数据结构，但用框架服务来管理文件。文件中包含有数据结构。因而，OMS 能建立文件和文件之间的连接，不能建立文件内部结构之间的连接。

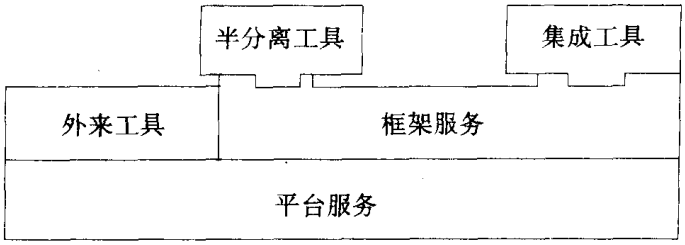


图11.22　软件工程环境的工具集成

③ 外来工具：这些工具仅用平台服务。它们管理自己的数据，但可能使用数据互换服务来把数据传进和传出。

如果已有工具与环境支撑部分运行在相同平台上，那么将已有工具作为外来工具和半分离工具移进软件工程环境就相对容易些。但是，工具和环境服务紧密集成的问题是一个“鸡与蛋”的问题。

11.3.5　PCTE

通用框架服务集的开发来源于 APSE 的石人提案。在美国和欧洲，已建立许多研究项目来定义通用工具接口。在美国，DoD 作了这方面的努力，集中在开发一个 Ada 环境的核心 APSE。最终开发出了通用 APSE 接口集即 CAIS(Common APSE Interface Set)。这原先是为了制定一个标准，不存在任何商业目的。

在美国 CAIS 项目进行的同时，欧洲委员会在 ESPRIT 研究项目下资助了一个多国项目来定义相对公共的工具接口。这就是 PCTE(可移植通用工具环境)，1984 年发布了 PCTE 的第一版。CAIS 标准是面向 Ada 的，与此不同，PCTE 标准是面向 UNIX 和 C 的。它旨在通用性而非支持面向语言的环境。

PCTE 有许多技术缺陷(例如，它缺乏对安全性和访问控制的支持)，过于紧密地与 UNIX 这一平台相连。为解决这些问题建立了一些项目进行第一步研究，其中包括由国防部门资助的来开发 PCTE＋的项目，以及由欧洲计算机行业协会(ECMA)资助的项目。ECMA PCTE 现在已取代了这些早期版本，并已作为标准被接受。

PCTE 和 CAIS 提案有许多重复交叉之处，因而要对这两者进行综合，开发一个称之为 PCIS(Portable Common Interface Standard)的标准(可移植通用接口标准)。在 1994 年初已发布 PCIS 框架定义的第一版。1995 年，它被原型化。

但是，欧洲和美国已广泛接受 ECMA PCTE。许多制造厂商如 Digital 和 IBM 已开发出该标准的实现。PCTE 的存在以及其实现和工具的开发，也许意味着 PCIS 绝不会真正出笼。PCTE 也许会成为软件工程环境框架的事实上的标准。

PCTE 的对象管理是基于实体-关系-属性(ERA)模型的，其对象与其它对象会存在关系，对象带有属性。PCTE 支持对象和对象间的连接。对象可分类，用户还能定义子对象。例如，从基本类型 TEXT，可能派生出像 C-SOURCE-TEXT 子类型。该类型允许工具检查其所操纵的对象是否具有正确类型，以减少错误的范围，使用带类型的连接来建立对象间的关系。在模式定义集(SDS)中定义类型，PCTE 中运行的任何程序都和一个工作模式相联，而工作模式由一个或多个 SDS 组成。

为提供数据恢复和复原，支持事务处理。一个事务是一个原子的动作集，其作用于数据之

上的方式为:或者执行全部动作,或者一个动作也不执行。如果在事务处理过程中发生错误,就可以恢复数据库到一个一致的状态。能管理事务的执行,支持进程间通信。允许启动、终止和控制进程,能将进程存储到OMS中,以便使用OMS查询机制找到进程信息。框架是可在工作站网络上运行的,因而支持进程和数据分派。

1992年后Brown等介绍了ECMA PCTE,并根据软件工程环境参考模型对它作了评估。如表11-5所示。

PCTE采用一个复杂的安全模型(在参考模型中未涉及)来控制OMS中对象的访问。提供了不同的安全级别,如机密,秘密等,这使得该框架有可能适应于军事应用。

表11-5 PCTE与软件工程环境参考模型的比较

服务	描述
数据仓库	除备份服务外,PCTE提供了所有的数据仓库服务
数据集成	除通用查询服务外,提供了所有的数据集成服务。像状态监控和数据互换服务等。比参考模型中提议的要少
任务管理	没有提供别的任务管理服务,除查账和记账服务外
消息	有消息分派服务但没有消息注册
用户界面	建议基于PCTE的环境都采用X-Window来实现其用户界面。没有强制要用哪些特别的库

表11-5的比较表明ECMA PCTE提供了一个相当完整的低级框架服务集,但是还需进一步扩充以解决软件工程环境参考模型提出的方方面面的问题。这可以通过在PCTE同级或之上实现别的服务来获得。例如,在美国DoD环境框架服务提案中,采用PCTE提供数据仓库和数据集成服务。控制服务由别的单独的系统提供,如由HP的SoftBench提供。用户界面服务由X/Motif提供,任务管理服务由Process Weaver提供。

11.4 大型软件开发环境青鸟系统简介

11.4.1 综述

"大型软件工程开发环境青鸟系统"是国家科技攻关课题成果,它是由北京大学牵头研制的一个面向对象的软件开发环境。

如前所述,国际上对于软件工程环境的研究与开发始于80年代初期。早期的软件工程环境是仅支持个别软件开发过程的单体型软件工程环境。到80年代中后期出现了把多种方法、技术和大量的工具集于一体、支持整个软件生存周期的集成型软件工程环境。90年代初期,随着面向对象方法发展到软件开发过程的各个阶段,出现了支持面向对象开发方法的集成型软件工程环境,但因缺乏实用的对象管理系统的支持,此类环境尚不够完备。

CASE是促进软件产业发展的重要因素。国外各大软件公司均有自己的软件工程环境作为提高竞争力的手段。我国政府和科技工作者多年来也十分重视软件工程环境的研究与开发。在"六五"期间,我国即安排了软件工程环境的攻关工作,北京大学在这方面的成果是"软件工程核心支撑环境BD-85"。"七五"期间的攻关研制成功了集成化软件工程环境青鸟Ⅰ型系统,该系统以环境信息库为核心,提供了支持软件生存周期各个过程的软件工具。在"八五"攻关期

间，成功地研制了青鸟Ⅱ型。本节主要介绍青鸟Ⅱ型，又称JB2。

JB2的特点：

(1) 把支持面向对象的软件开发，作为环境的主要目标之一，具体体现为：

① 研制和开发作为集成型软件工程环境核心的对象管理系统JB2/OMS。

② 以对象管理系统为核心的JB2环境的总体结构。

③ 设计并实现支持永久对象的面向对象编程语言CASE-C++。

④ 在对象管理系统和CASE-C++语言的支持下进行环境中大部分软件工具的开发和集成。

⑤ 提供一套支持面向对象分析、设计和编程的OO系列工具。

⑥ 为支持图形用户界面的面向对象开发，提供一个包括大量界面常用成分的界面类库和一个面向对象的界面辅助生成器。

总之，通过对象管理系统、CASE-C++语言、OO系列工具、界面类库及界面辅助生成器的研制，使环境对工具设计者和最终用户的面向对象软件开发形成强有力的支持。

(2) 较为成功地研制了以数据集成、控制集成和界面集成为中心的开放性环境集成机制，包括：

① 以对象管理系统为核心的数据集成部件。

② 以消息服务器和过程控制系统作为控制集成部件。

③ 以基本OSF/Motif的界面类库和界面辅助生成器作为界面集成部件。

④ 在环境集成机制中提供开放性的工具插槽。

⑤ 制定符合开放性要求的工具结构模型。

(3) 支持多种软件开发方法，包括结构化方法，信息模型法和OO方法。

(4) 系统既可以集成支持软件生存周期全过程的软件工具，成为通用性软件工程环境，又可以通过剪裁而形成支持特定应用领域的专用性应用开发平台。

11.4.2 JB2系统介绍

下面首先介绍JB2的总体结构，然后介绍JB2环境集成机制和JB2环境中的软件工具。

1. JB2总体结构

JB2是一个面向对象的集成化软件工程开发环境，系统本身的开发也较全面地采用了面向对象的技术。在设计上，JB2参考国际上一些著名的环境模型，通过对它们的分析和评估，吸取其优点，提出了基于自己已有研究成果之上的集成环境模型。

JB2集成机制体现了数据、控制和界面三方面的集成，其数据集成基于对象管理系统(OMS)，控制集成基于消息服务器，界面集成基于OSF/Motif以及在此基础上开发的JB2界面类库和界面辅助生成器。同时，环境中提供了一个面向永久对象的编程语言CASE-C++作为工具的书写语言，保证了工具在环境中的紧密集成。

JB2的集成机制形成了一个开放的工具插槽。工具的设计遵照统一的工具结构模型，并提倡以CASE-C++语言编写，从而很容易集成到环境中并达到较高的集成度。外来工具在按规范要求封装后也容易被环境接纳。工具之间的通信由消息服务器完成。

图11.23显示了JB2的总体结构。

JB2环境的开放的工具插槽保证了环境易于剪裁和扩充。按结构模型规范要求封装好的

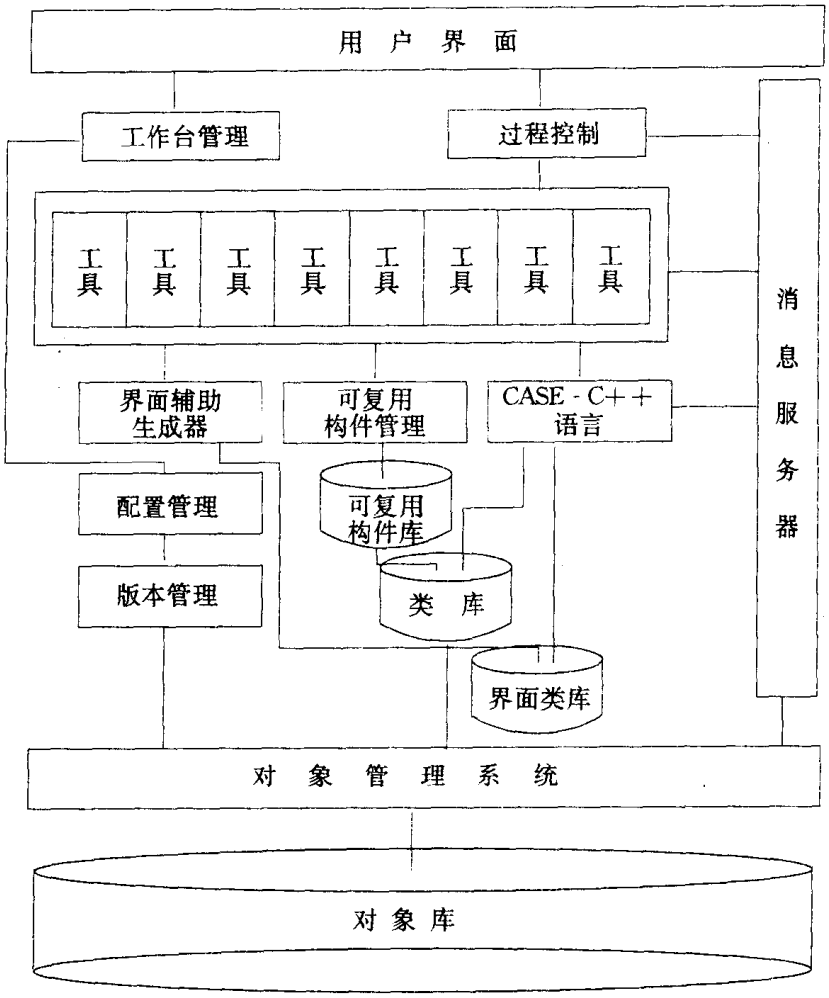


图11.23 JB2系统的总体结构

工具，实质上构成了一个“插件”，工具向环境的集成仅是将“插件”插入工具插槽中，工具从环境中删除只需将“插件”从插槽中“拔出”。JB2 集成机制所体现出的插槽形式为环境的扩充和剪裁提供了很大方便，也保证了集成和开放的有机统一。图 11.24 显示了工具“插件”与环境工具插槽间的基本关系。

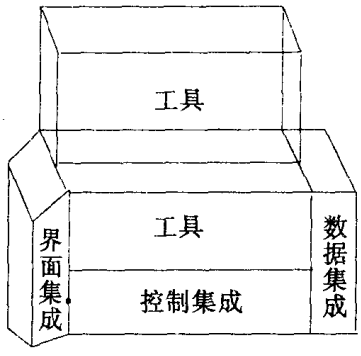


图11.24 JB2工具插槽

2. 环境集成机制的主要部件

JB2 环境集成机制的主要部件如下：

(1) 对象管理系统

JB2 的对象管理系统(JB2/OMS)是 JB2 环境的核心，其作用是对软件开发过程中定义的对象类以及对象实例进行存储和管理。JB2/OMS 实现了永久对象的存储与管理，可并发地运行于网络中各个工作站之上，支持在多用户、多程序之间对象的共享和并发执行，由并发控制机制保证其属性信息的一致性。JB2/OMS 对于环境的工具开发者和最终用户提供了统一的面向对象技术支持，即，它管理和存储的对象，既包括构成工具本身的对象，也包括用户在使用工具时产生的结果对象，以及用户以 CASE-C++语言开发的任何应用程序中的对象。JB2/OMS 分为以下几个主要部分：

① 类库：JB2 的类库保存并管理环境中长期使用的全部类定义以及类之间的继承关系。库中的类包括：由环境提供的一组预定义的类，各工具开发者提供的工具成分类，一般用户在软件开发过程中不断提交的类。此外，JB2 为支持基于 OSF/Motif 的图形用户界面的开发，提供了一个丰富而高效的界面类库，它是整个 JB2 类库的一个相对独立的子库。类库的主要意义是支持类定义在不同用户、不同程序间的共享和复用，并进行权限控制。

② 对象库：JB2/OMS 的对象库保存并管理 JB2 环境中的全部永久对象的实例信息。这个对象库是 CASE-C++语言实现永久对象概念的基层支持。由于这种支持，使用户(包括工具开发者和一般用户)在程序中声明的永久对象能够超越程序运行时间而长期存在。此外，它使程序员可以把永久对象看作是“无缝的”，而不必关心它在内外存之间的转换。JB2/OMS 的设计采用了 Client/Server 结构，使库中的对象可以在网络中并发执行的各个程序或进程之间共享，为了保证对象的数据完整性，OMS 设立了锁管理机制。

③ OMS 浏览器：OMS 浏览器是 OMS 的交互式浏览及维护界面，它允许用户浏览类库和对象库中的全部可见的类及对象以及它们之间的结构关系。用户还可以通过它交互式地定义新类、修改类的权限或修改处于开发状态的类定义。环境的特权用户除了可使用上述功能之外，还可以删除已经无用的对象实例和对象类。

(2) CASE-C++语言

CASE-C++语言是 JB2 环境中的一个面向对象的编程语言(OOPL)。设计和实现这个语言的主要动机是为了提高在软件工程环境下面向对象的软件开发工作的效率并达到软件工具在环境中的紧密集成，而现有 OOPL 在许多方面不能很理想地达到上述要求，如对永久对象的支持。JB2 选择了目前应用比较广泛的 C++语言作为基础，并针对上述要求加以扩充，命名为 CASE-C++语言。它对 C++完全兼容，主要有以下几点扩充：

① 支持永久对象

CASE-C++语言在 JB2 的对象管理系统(JB2/OMS)支持下提供了永久对象(Persistent Object)的定义和处理能力。用户在程序中创建一个对象时，可以用保留字 persistent 声明它是永久的。这样的对象的生命期可超越程序运行的时间而在 OMS 的对象库中长期保存。本程序或其它程序再次使用这个对象时，对象呈现它上一次使用时的状态。永久对象的描述及处理机制使用户可以把对象看作是“无缝的”——程序员不必关心每个对象是在内存还是在外存。对象在内、外存之间的转储，是由 CASE-C++在必要时自动地调用 OMS 的基本读写函数实现的。程序员不必借用其它存储管理系统来保存对象，从而解除了在程序中进行数据转换和显式

地存储与恢复所带来的负担。

② 对象之间关系的强化描述及永久存储——链(Link)机制的引入

CASE-C++引入了链的概念和处理机制来表示对象之间的关系。一个链,从一个源对象出发,链接到一个目的对象。它作为对象的一个属性(实例变量)在源对象中定义,它的值则是目的对象的永久性标识。链机制可使对象之间的关系与对象一起得到长期的保存,这是链和非永久性 OOPL 中"对象指针"概念的重要区别。链的另一个特点是它可以带有自己的链属性,以描述较复杂的关系信息。因此,在 CASE-C++中对象关系的描述能力大大增强。

③ 支持类定义的复用和共享

CASE-C++语言提供了使类定义成为可复用构件并提供其它用户共享的设施。在 CASE-C++程序中定义的类,可以用 export 语句提交到类库,以供复用和共享;另一方面,程序中可以用 import 语句从类库中引入自己所需要(并且有权使用)的类。

CASE-C++语言允许 C++语言原有的类定义形式(以小写 class 为保留字),同时增加了扩充的类定义形式(以大写的 CLASS 为保留字),凡属以下几种情况必须使用扩充的类定义形式。

- 类定义的内部使用了 CASE-C++的扩充语法成分(例如链);
- 准备移出以提供复用和共享的类;
- 准备创建永久对象的类。

④ 可变长属性的描述——对象内容

CASE-C++以"对象内容"作为对象的一种属性,它的长度是可变的。这一扩充使对象的属性可以描述那些长度动态变化的数据信息,例如,进行交互式编辑的图形、正文信息。在软件工具中,被加工和输入/输出的信息多属此类。因此,这一扩充特别适应软件工具的开发。

(3) 用户界面、界面类库和界面辅助生成器

JB2 的用户界面是用 OSF/Motif 开发的。为了从根本上提高用户界面的编程效率,并保证界面风格的一致性,我们在 OSF/Motif 之上开发了一个 JB2 界面类库。界面类库是 JB2 总类库的一个独立的子库,库中的类是针对各种常用的界面部件定义的界面对象类。每个类具有 C++和 CASE-C++两种语言的定义形式,均用 Motif 实现。在 JB2 界面类库的支持下,程序员要定义一个界面成分时,只需引用相应的界面类创建一个具有指定属性值的对象。此外,可用类中提供的函数来控制界面的变化。这使得界面部分的编程工作大为简化(典型地,百余行的 Motif 程序可以简化到一行到几行 C++或 CASE-C++程序)。另一方面,它也使不熟悉 Motif 的程序员能够快速地胜任界面开发工作。

界面类库中预定义的界面类较全面地包括了一般用户常用的界面成分。用户还可以利用这些预定义的类通过派生或组合产生自己的新类并提交到类库。

界面辅助生成器是 JB2 环境中的一个交互式界面生成工具,它为用户的界面开发提供了可视化的支持。用户可以在屏幕上以"所见即所得"的方式构造自己所需要的界面成分。最终辅助生成器将把屏幕上的开发结果转换为 C,C++,CASE-C++或 UIL 程序源代码。界面辅助生成器既可看成环境集成机制的一个组成部分,也可以作为一个独立的软件工具。

JB2 的界面类库和界面辅助生成器可以有效地保证界面设计的规范化和风格的一致性,并且明显地提高了界面的开发效率和质量。

综上所述,JB2 的界面开发支持层次如图 11.25 所示。

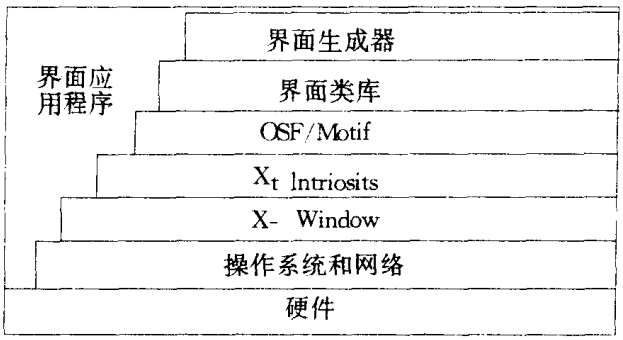


图11.25 JB2界面支持层次

(4) 可复用构件库

JB2 可复用构件库的设计目标是为环境提供一个支持多种复用级别的软件复用机制。源程序一级的复用主要在 JB2/OMS 中的类库和 JB2 界面类库的基础上实现。其它级别的复用还包括分析构件和设计构件的复用。对象化的构件是库中最规范的可复用构件,其中包括在 OOA 阶段产生的分析类,OOD 阶段产生的设计类和用 OOPL 定义的编程类;同时库中也将包括按传统方法开发并按构件库的描述规范所描述的非对象化构件。

JB2 构件库的组织允许表示可复用构件之间的多层关系,例如对象化构件之间的继承关系、组成同一系统的构件之间的组合关系、从分析构件到设计构件到编程构件之间的演化关系,以及应用领域分类关系等等。

JB2 可复用构件库以基于关键词的检索和交互式的提问细化和理解、定位作为重要的检索策略、并在多层关系模型的支持下提高检索效率。

(5) 消息服务器

JB2 的消息服务器是在 UNIX 系统的 IPC/RPC 基础上开发的一个高层次的消息服务设施,它用于工具或环境其它独立部件之间的通信服务。消息的发送和接收单位是工具(或环境部件)。消息是网络透明的,其语义由接收单位和开发单位相互约定,并按 JB2 的统一约定进行描述和登记。

消息服务器是提高环境的控制集成度的重要设施,它为相关工具之间的配合工具和切换提供了方便而有效的支持。例如,设计工具在工作时可以发送消息要求分析工具显示相关的分析文档并进行修改,而操作员不必在这两个工具之间频繁地退出和进入。用JB2 消息服务器实现工具之间的通信比直接使用 UNIX 系统的 IPC/RPC 要简练得多,例如程序员不必关心事件队列的定义、存取和查询等细节。

消息服务器也是 JB2 过程控制基础,它为过程控制的实现提供了通信服务。

(6) 过程控制

过程控制是提高软件质量及生产率的重要保证。对过程模型及其描述语言以及过程驱动软件工程环境的研究是当前 CASE 研究的一个热点。JB2 中探讨了一个面向对象的通用过程模型 OOSP,在该模型中,软件开发过程由一系列对象组成,对象间的交互构成了软件开发的活动。相应地,提出一个面向对象的过程描述语言 OOPDL,使用该语言可以灵活地描述并控制机制建立在消息服务器之上,在它的支持下开发者可以方便地运行软件过程,以提高软件生产率及软件质量。

(7) 配置管理、版本管理和工作台管理

配置管理、版本管理和工作台管理是当前真正受用户欢迎的软件工程环境中都应具备的功能部件。

配置管理系统通过它的配置库保存用户在环境支持下产生的各种对象，支持这些对象的共享和权限管理，并实现由单元对象(或者系统对象)合成为系统对象的基本操作。

版本管理系统保存配置库中每个对象的各种版本及版本之间的演化关系，实施权限控制，并把版本按使用工作台的用户的要求提供给用户。

在对象管理系统的支持下，配置库和版本管理系统只需保持对象及其版本的逻辑映像，其物理映像由对象库存储，减轻了实现的难度，同时保证了较高的数据集成度。

JB2 中工作台的概念和我们前面介绍的工作台概念不同，在 JB2 中，工作台是环境用户完成一组工作的专用场所。环境为每个用户提供一个树形的工作台结构，用户可以在此结构上创建多个子工作台；每个工作台的作用是作为用户选择工具或使用环境设施(如配置管理、类库、可复用构件库等)的控制界面；作为工具与环境的数据集成机制之间的中转站，工具的输出对象在正式提交配置库之前以及从配置库提取出来准备加工的对象，都临时存放在工作台上；提供对工作台上所存放的对象的选取、复制、删除、提交等操作。

总之，①应该充分发挥 OMS 的作用，上层部件不再直接存放对象的物理实体，而是存放和管理它们的逻辑映像。②具有多个版本的对象，由版本管理部分管理其版本结构，但每个版本的实体被看作 OMS 中的一个独立的对象。这种处理方法既避免了版本管理部件的物理存储负担，又减少了 OMS 设计的逻辑复杂性。③工作台管理体现了界面集成和数据集成两个方面。首先，环境用户是在统一的工作台界面上工作，通过工作台界面，用户可以进一步启动工具或进行配置管理、版本管理；其次，JB2 中的工作台提供了环境用户的工作空间和私有数据存储空间，存储用户在开发软件过程中所产生的而尚未提交的对象，以及对配置库中永久对象的加工都在这里进行。

3. 工具结构模型和环境中的工具

(1) 工具结构模型

工具与环境的接口问题是关系到环境的集成度和开放性的关键问题之一。在没有任何约定的情况下开发出来的工具很难在环境中达到紧密的集成。单纯依靠小范围内的特殊约定进行工具与集成机制的开发又很难使环境具有开放性。在统一的国际标准形成之前，解决这一问题的可行办法是寻找一个对环境适应性较强、开发者容易实施的工具结构模型。青鸟Ⅱ型采用如图 11.26 所示的工具模型。青鸟Ⅱ型系统中 20 多个工具的开发单位使用这个模型来解决工具与环境的接口问题。

在这个模型中，一个工具由四个独立部件构成，即功能部件、数据接口部件、控制接口部件和界面接口部件。其中，功能部件是完成工具自身的功能所需的软件成分，其它三个部件分别实现工具与环境的数据接口、控制接口和界面接口。所有与环境接口有关的设计决策都集中体现在这三个接口部件中，从而隔离了环境对功能部件设计的影响，使工具开发者的大部分工作(有关功能部件的工作)可以独立于环境。按这一模型设计的工具，相当于环境中的一个标准插件，三个接口部件是数据、控制和界面三方面的“插头”，环境的数据集成、控制集成和界面集成机制则是与之对应的“插槽”。

这一模型对工具和环境集成机制具有双重的意义。对工具而言，由于功能部件的独立性，

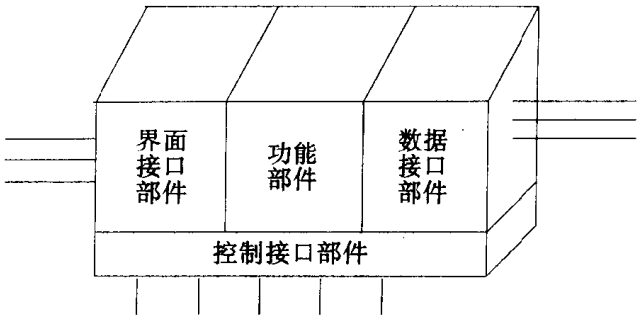


图11.26 JB2工具结构模型

当需要进入一个新的环境时，只需要更换或修改它的接口部件，这使得工具对不同的环境具有较强的可移植性。对环境集成机制而言，一旦确定了“插槽”的规格，其设计则不受具体工具的影响，从而可按照统一的工具结构模式去追求较高的集成度。集成工作也大为简化，因为不必针对每个具体的工具去解决接口问题。此外，按照这个工具结构模型对“外来工具”进行改造或“封装”——使之具有符合环境要求的接口部件，如同给一个非标准的插件装上标准的插头——可以在很大程度上提高环境的开放性。

(2) JB2 环境中的工具

JB2 环境中的工具分为三大类，即①传统类工具，覆盖整个软件生存周期，支持传统的软件开发方法；②OO 工具，包括从 OOA 到 OOP 的一系列工具；③应用类工具，支持特定应用领域的应用软件开发。

工具举例如下：

- 结构化分析工具 SAT；
- 需求文档分析工具 DAT/DFD；
- 结构化设计工具 SDT；
- 设计文档分析工具 DAT/MSD；
- 文档追踪工具 TRACE；
- 详细设计工具 DDT；
- 详细设计分析工具 DAT/PDL；
- C 编码工具 CCT；
- PAD 图到 C 转换工具 PADT；
- C 程序测试工具 CSTT；
- Fortran 程序测试工具 FSTT；
- C 程序维护工具 SMT；
- 软件项目管理工具 SPMT；
- 软件价格模型估算工具 CMET；
- 用户界面生成工具 UIGT；
- 用户界面生成工具 UIGS；
- 用户界面生成工具 GUIDS；
- 面向对象分析工具 OOAT；
- 面向对象设计编程工具 OODPT；
- 数据库设计工具集 DBTOOLS；

- 数据库应用生成工具 MISGT；
- 地理信息系统辅助工具集 GIS；
- 实时监控系统辅助生成工具 SCADA；
- 人工神经网辅助工具 NNET；
- 网络应用辅助生成工具 RODA；
- 分布式系统辅助生成工具集 DSGT；
- 文档出版工具 DPT。

4. JB2 环境的剪裁——支持不同开发方法和应用领域的软件开发平台

JB2 环境的开放性集成机制和 JB2 工具结构模型使环境具有良好的可剪裁性。根据具体应用的需要对环境进行剪裁，可产生以下几个软件开发平台：

(1) 支持结构化方法的开发平台 JB2/B(中西文版)

该平台含有结构化开发方法有关的环境部件和软件工具，支持瀑布模型中的各个软件过程。

(2) 支持 OO 方法的开发平台 JB2/OO(中西文版)

该平台含有与 OO 开发方法有关的环境部件、软件工具语言。

(3) 地理信息系统开发平台 JB2/GIS(中文版)

该平台含有面向对象的分析、设计与编程工具和地理信息系统专用工具集、支持对地理信息系统进行面向对象开发。

(4) MIS 开发平台 JB2/MIS(中文版)

该平台含有支持结构化方法和 E-R 模型的前期工具和管理信息系统的专用开发工具，支持 MIS 的开发。

(5) 实时监控系统开发平台 JB2/SCADA(西文版)

该平台含有结构化开发方法的一般工具和实时监控系统的专用工具，支持实时监控系统的开发。

106936

参考文献

[1] 杨芙清主编. 计算机科学与技术百科全书软件分支. 清华大学出版社(即将出版)

[2] 杨芙清,邵维忠,梅宏. 面向对象的 CASE 环境青鸟Ⅱ型系统的设计与实现. 中国科学,A 辑 1995:533—542

[3] Jag Sodhi. Softuare Eugineering Methods, Management, and CASE Tools. Mcgvaw-Hill. Inc. 1991

[4] 冯玉琳,等. 软件工程. 中国科学技术大学出版社,1992

[5] 邵维忠,廖钢城,李力译,杨芙清校. 面向对象的分析. 北京大学出版社,1991

[6] 邵维忠,廖钢城,苏渭珍译,杨芙清校. 面向对象的设计. 北京大学出版社,1994

[7] 唐世渭,方裕译;徐家福,杨芙清校. 软件工程——实践者的研究途径和方法. 小型微型计算机系统,1984

[8] 徐家福,王志坚,翟成祥. 对象式程序设计语言. 南京大学出版社,1992

[9] 李健. 软件过程建模技术和过程实施技术研究. 博士论文,1995

[10] Owicki, S and Gries, D. An axiomatic proof technique for parallel programs. Acta Informatica 1976:319—340

[11] Hoare, C. A. R. An axiomatic basis for computer programming. CACM 12, 1969: 576—580,583

[12] 北京大学计算机科学技术系 CASE 研究室技术报告. 集成化软件工程支撑环境 JB2, 1991

[13] Lan Sommerville. Softuare Engineering. Addison-Wesley, 1996

[14] 蔡希尧,陈平. 面向对象技术. 西安电子科技大学出版社,1993